МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 50.00.00.000 ПЗ

Група ІПм-24-1

Герасимчук Андрій

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Герасимчук Андрій Сергійович
(прізвище, ім'я, по батькові)

УДК 004.344
(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми інтеграцій чат-ботів у веб-додатки

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Герасимчук А. С.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Ксенич Андрій Іванович, к.т.н. доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту
Завідувач кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль
доц. Вовк Р.Б.
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2025 р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Герасимчуку Андрію Сергійовичу
(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “Моделі, методи та алгоритми інтеграції чат-ботів у веб-додатки”

керівник проекту (роботи) Ксенич Андрій Іванович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 05 ” листопада 2025 р. № 695/7

2. Строк подання студентом проекту (роботи) 15 грудня 2025 р.

3. Вихідні дані до проекту (роботи) Код серверу, алгоритми роботи з оркестрацією, код контейнеризації, алгоритм роботи з ШІ

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Теоретичний аналіз технологій та методів побудови і інтеграції чат-ботів
2. Дослідження архітектурних підходів та методів проєктування систем для чат-ботів
3. Проєктування моделей, методів та алгоритмів інтеграції чат-ботів у платформу
4. Програмна реалізація та впровадження системи

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Зростання задоволеності клієнтів завдяки впровадженю AI в чат-ботах (рис. 1.7, ст. 25)
2. Графік зростання складності в впровадженні автономних ботів (рис. 2.3, ст. 36)
3. Графік залежності часув повного зчитування (рис. 2.5, ст. 43)
4. Схема взаємодії самостійних ботів з інтеграційним шаром (рис. 3.4, ст. 56)
5. Схема розгортання та взаємодії контейнеризованих чатботів (рис. 3.5, ст. 57)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.	

7. Дата видачі завдання 04 вересня 2025 р.

Керівник ______________________________
(підпис)

Завдання прийняв до виконання______________________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури	20.09.2025	виконано
2	Теоретичний аналіз технологій та методів побудови й інтеграцій чат-ботів	05.10.2025	виконано
3	Дослідження архітектурних підходів: інтеграційний шар, зберігання повідомлень, підходи до AI-інтеграції	12.10.2025	виконано
4	Проєктування моделей та алгоритмів до інтеграції чат-ботів у платформу	25.10.2025	виконано
5	Формулювання вимог та опис архітектури розробленої платформи	25.11.2025	виконано
6	Програмна реалізація рішення	05.12.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр ____________________
(підпис)

Керівник роботи ____________________
(підпис)

АНОТАЦІЯ

Магістерська робота: 107 с., 36 рис., 3 табл., 0 дод., 40 джерел.

Тема: Моделі, методи та алгоритми інтеграції чат-ботів у веб-додатки.

Об’єкт дослідження: процес інтеграції та взаємодії веб-додатків із зовнішніми платформами обміну повідомленнями.

Мета роботи: підвищення ефективності функціонування та управління діалоговими системами шляхом розробки уніфікованих моделей та алгоритмів інтеграції чат-ботів у веб-додатки.

Предмет дослідження: моделі, методи та алгоритми уніфікації обробки даних від чат-ботів різних архітектур та організація їх централізованого управління.

Результати дослідження:

Виконано аналіз підходів до побудови багатоканальних систем чат-ботів і запропоновано удосконалений метод інтеграції, що базується на адаптивній абстракції для нормалізації даних від різних платформ та гібридній обробці запитів із використанням моделей штучного інтелекту і збереженням контексту розмови в розподіленому середовищі. Запропонований підхід забезпечує кращу масштабованість та спрощує підключення нових платформ без суттєвих змін в системі.

Висновок:

У результаті виконання магістерської роботи розроблено та впроваджено програмне рішення, що забезпечує уніфіковану інтеграцію чат-ботів із веб-додатками, централізоване керування їх конфігураціями та історією взаємодії, а також підтримує використання AI-компонента для формування відповідей.

ЧАТ-БОТИ, ВЕБ-ДОДАТКИ, ІНТЕГРАЦІЯ, SaaS-ПЛАТФОРМА, УНІФІКАЦІЯ ПОВІДОМЛЕНЬ, МЕСЕНДЖЕРИ, API, МАРШРУТИЗАЦІЯ ПОВІДОМЛЕНЬ, КОНТЕКСТ ДІАЛОГУ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ВЕЛИКІ МОВНІ МОДЕЛІ.

ANNOTATION

Master's thesis: 107 p., 36 fig., 3 tab., 40 sources.

Topic: Models, methods, and algorithms for integrating chatbots into web applications.

Object of research: the process of integration and interaction of web applications with external messaging platforms.

Purpose: to improve the efficiency of operation and management of dialogue systems by developing unified models and algorithms for integrating chatbots into web applications.

Subject of research: models, methods, and algorithms for unifying data processing from chatbots of different architectures and organizing their centralized management.

Research results:

An analysis of approaches to building multi-channel chatbot systems was performed, and an improved integration method was proposed. The method is based on an adaptive abstraction for normalizing data from different platforms and hybrid request processing using artificial intelligence models while preserving the conversation context in a distributed environment.

Conclusion:

As a result of this master's thesis, a software solution was developed and implemented that provides unified integration of chatbots with web applications, centralized management of their configurations and interaction history, and also supports the use of an AI component for generating responses. The proposed approach provides better scalability and simplifies the connection of new platforms without significant changes to the system.

CHATBOTS, WEB APPLICATIONS, INTEGRATION, SaaS PLATFORM, MESSAGE UNIFICATION, MESSENGERS, API, MESSAGE ROUTING, DIALOGUE CONTEXT, MICROSERVICE ARCHITECTURE, LARGE LANGUAGE MODELS.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ... 8

ВСТУП..9

РОЗДІЛ 1

ТЕОРЕТИЧНИЙ АНАЛІЗ ТЕХНОЛОГІЙ ТА МЕТОДІВ ПОБУДОВИ І ІНТЕГРАЦІЇ ЧАТ БОТІВ.. 14

1.1 Визначення та класифікація чат-ботів... 14

1.2 Аналіз архітектурних моделей взаємодії чат-ботів з платформами..................18

1.3 Використання штучного інтелекту для автоматизації діалогів.......................... 24

Висновок до розділу... 29

РОЗДІЛ 2

ДОСЛІДЖЕННЯ АРХІТЕКТУРНИХ ПІДХОДІВ ТА МЕТОДІВ ПРОЄКТУВАННЯ СИСТЕМ ДЛЯ ЧАТ-БОТІВ.. 31

2.1 Методика побудови багатоплатформенного інтеграційного шару....................31

2.2 Дослідження архітектури бази даних для зберігання повідомлень чат-ботів.. 38

2.3 Методи інтеграції моделей штучного інтелекту... 45

Висновок до розділу... 48

РОЗДІЛ 3

ПРОЄКТУВАННЯ МОДЕЛЕЙ, МЕТОДІВ ТА АЛГОРИТМІВ ІНТЕГРАЦІЇ ЧАТ-БОТІВ У ПЛАТФОРМУ...50

3.1 Підходи до створення та підключення чат-ботів... 50

3.2 Проєктування структури бази даних в системі чатботів...................................59

3.3 Моделі організації черг та обробки подій на основі RabbitMQ........................ 68

3.4 Архітектурні рішення інтеграції LLM.. 72

Висновок до розділу... 77

РОЗДІЛ 4

ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ....79

4.1 Архітектура розробленої SaaS-платформи....79

4.2 Реалізація модулів управління ботами.... 86

4.3 Реалізація AI-компонента.... 92

4.4 Демонстрація роботи веб-додатка та користувацьких сценаріїв.... 97

Висновок до розділу....100

ВИСНОВКИ....102

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....104

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface.

SSL – Secure Sockets Layer

NLP – Natural Language Processing

AI – Artificial Intelligence

CSAT – Customer Satisfaction Score

AMQP – Advanced Message Queuing Protocol

CRM – Customer Relationship Management

HTTP – HyperText Transfer Protocol

JSONB – Binary JSON

LLM – Large Language Model

NACK – Negative Acknowledgement

ORM – Object-Relational Mapping

payload – Корисне навантаження повідомлення/події

REST – Representational State Transfer

SaaS – Software as a Service

UI – User Interface

UX – User Experience

UUID – Universally Unique Identifier

ВСТУП

Актуальність роботи

В сучасних умовах швидкого розвитку та цифровізації бізнес-процесів месенджери з простих засобів комунікації перетворюються в дійсно потужні інструменти взаємодії з клієнтами. Для багатьох компаній месенджери стали основним каналом надання оперативних послуг, інформації та клієнтської підтримки. Це поступово витіснило традиційні методи зв’язку між клієнтом та бізнесом. Однак, зі зростанням обсягів запитів та кількості чатів, потреба у цілодобовій доступності вимагають переходу до більш автоматизованих рішень. Тоді на перший план виходять чат-боти – систем, які здатні вести діалог природною, зрозумілою людині, мовою та автоматично виконувати рутинні завдання.

Проте процес розробки та інтеграції таких систем в вебсередовище супроводжується низкою системних та технічних проблем, пов’язаних з імплементацією та фрагментацією технологій. Попри спільне призначення, реалізація чат-ботів в різних платформах суттєво відрізняється. Кожна платформа має власний унікальний програмний інтерфейс, специфічні формати представлення даних та відмінні архітектурні принципи взаємодії, чи то використанння веб-хуків або методів постійного опитування сервера. Така різнорідність призводить до того, що рішення, розроблене для однієї платформи, є несумісним іншою без значних модифікацій програмного коду.

Сучасний ринок потребує не просто ізольованих рішень, а комплексних систем управління, які дозволяють централізовано налаштовувати логіку роботи ботів, збирати аналітику та інтегрувати їх вже з існуючими веб-платформами. Окремою складністю стає те, що від таких систем очікують дедалі “розумнішої” поведінки. Тепер уже недостатньо простих заздалегідь прописаних сценаріїв – потрібні інтеграції з моделями генеративного штучного інтелекту, які здатні самостійно формувати змістовні та доречні відповіді.

Ефективна інтеграція подібних ботів вимагає не лише впровадження та інтеграції окремих технологій, а створення надійної архітектури, яка забезпечить

стандартизоване та безпечне вбудовування чат-ботів в веб-додаток. Така систематизація дозволить мінімізувати технічні бар'єри, оптимізувати витрати підприємств на ту чи іншу оптимізацію комунікацій та забезпечити інноваційний підхід до клієнтського сервісу.

Тому розробка уніфікованих моделей, методів та алгоритмів, які дозволяють абстрагуватися від технічних особливостей конкретних месенджерів та інтегрувати різнорідні чат-боти у єдину екосистему на основі моделі програмного забезпечення як послуги, є актуальним науково-прикладним завданням.

Порівняння роботи з відомими розв'язаннями проблеми

Відомі підходи до побудови чат-ботів для веб-середовища та месенджерів здебільшого зосереджені або на використанні готових платформ і конструкторів, або на застосуванні фреймворків, що забезпечують підключення каналів комунікації та реалізацію сценаріїв через webhooks і серверні модулі. Зокрема, промислові рішення на кшталт Microsoft Bot Framework передбачають підключення різних каналів і обмін повідомленнями через стандартизовані REST-інтерфейси та конекторний шар, а Dialogflow пропонує механізм fulfillment, у межах якого бізнес-логіка реалізується у власному webhook-сервісі.

Запропонована в даній роботі система відрізняється від наведених рішень тим, що орієнтована на уніфіковану SaaS-архітектуру керування багатоплатформенними ботами з централізованими налаштуваннями, зберіганням даних і моніторингом, а також з провайдер-незалежним шаром підключення LLM. Це дозволяє змінювати або комбінувати моделі без прив'язки до конкретного вендора і, за потреби, підсилювати якість відповідей через підходи на кшталт Retrieval-Augmented Generation для роботи з доменними знаннями та обмеженнями контекстного вікна.

Мета і задачі дослідження

Метою магістерської роботи є підвищення ефективності функціонування та управління діалоговими системами шляхом розробки уніфікованих моделей та алгоритмів інтеграції чат-ботів у веб-додатки.

Розроблювана модель повинна забезпечувати уніфіковане представлення повідомлень і подій незалежно від провайдера, підтримувати централізоване керування конфігураціями ботів та інтеграцію з LLM, а також гарантувати коректну маршрутизацію звернень у багатокористувацькому середовищі з відновленням контексту діалогу. Додатковими вимогами є масштабованість сервісної архітектури, можливість розширення переліку платформ і провайдерів без зміни ядра бізнес-логіки, а також наявність інструментів моніторингу й оцінювання якості роботи системи.

Досягнення поставленої мети включало розв’язання таких **задач**:

1) провести аналіз архітектурних особливостей програмних інтерфейсів популярних месенджерів та дослідити існуючі підходи до побудови систем управлінням чат-ботами;

2) розробити формальну модель уніфікованого повідомлення та метод нормалізації вхідних джерел для їх подальшої обробки єдиною бізнес-логікою;

3) спроєктувати архітектуру програмного комплексу, що підтримує роботу з різними провайдерами ботів та забезпечує інтеграцію моделей штучного інтелекту;

4) розробити алгоритми маршрутизації повідомлень у багатокористувацькому середовищі та методи керування станом діалогу для забезпечення неперервності комунікації;

5) виконати програмну реалізацію платформи з використанням сучасних технологій та підходів, провести тестування ефективності розробленого рішення.

Об’єктом дослідження є процес інтеграції та взаємодії веб-додатків із зовнішніми платформами обміну повідомленнями.

Предметом дослідження є моделі, методи та алгоритми уніфікації обробки даних від чат-ботів різних архітектур та організація їх централізованого управління.

Методи дослідження

У роботі використано методи системного аналізу для порівняння функціональних можливостей месенджерів та визначення вимог до системи; методи об'єктно-орієнтованого проектування та патерни проектування для розробки архітектури уніфікованого шлюзу; теорію алгоритмів для створення логіки обробки та маршрутизації повідомлень; методи реляційної алгебри для проектування схеми бази даних.

Наукова новизна одержаних результатів

Удосконалено метод інтеграції чат-ботів, який на відміну від існуючих, використовує адаптивну абстракцію для нормалізації структур даних від різних платформ, що дозволяє обробляти різнорідні запити в межах єдиного сервісу незалежно від способу їх отримання. Впровадження гібридного методу обробки запитів з використанням моделей штучного інтелекту та збереженням контексту розмови в розподіленому середовищі.

Практичне значення одержаних результатів

Спроєктовано та програмно реалізовано платформу для створення та управління чат-ботами, яка дозволяє бізнесу підключати чат-ботів з різних месенджерів, налаштовувати логіку відповідей та використовувати в них штучний інтелект без необхідності написання програмного коду. Використана архітектура дозволяє легко збільшувати масштаби використання системи та спрощує додавання нових каналів комунікації.

Особистий внесок студента

1. Виконаний автором аналіз предметної області та існуючих підходів до інтеграції чат-ботів у веб-додатки і багатоканальні середовища месенджерів із формуванням вимог до уніфікованої платформи керування.

2. Проведено аналіз для оптимізації та уніфікації інтерфейсів та систем взаємодії.

3. Виконано дослідження в новітньому напряму в сфері ШІ.

4. Запропоновані та реалізовані автором архітектурні рішення, модель уніфікованого повідомлення, схема бази даних і алгоритми обробки та маршрутизації подій, а також програмна реалізація платформи з інтеграцією LLM і підтримкою взаємодії з різними провайдерами чат-ботів.

Структура магістерської роботи

Магістерська робота викладена на 108 сторінках друкованого тексту, який складається з вступу, чотирьох розділів, висновків, списку використаних джерел(40 найменувань). Робота містить 36 рисунків.

РОЗДІЛ 1
ТЕОРЕТИЧНИЙ АНАЛІЗ ТЕХНОЛОГІЙ ТА МЕТОДІВ ПОБУДОВИ І ІНТЕГРАЦІЇ ЧАТ БОТІВ

1.1 Визначення та класифікація чат-ботів

1.1.1 Сутність та ключові функції діалогових систем

Розвиток чат-ботів(інакше – діалогові системи) відбувся паралельно з еволюцією цифрових технологій, і поступово ці системи перетворились з простих програмних механізмів на повноцінні інтелектуальні платформи взаємодії між людиною й комп'ютером. Сучасні чат-боти – це не лише простий інструмент автоматизації окремих операцій, а універсальний засіб організації комунікації в середовищах, де потрібна швидка, структурована та постійна доступна форма обміну інформацією.

У своїй основі чат-боти є програмною системою, яка здатна аналізувати вхідні повідомлення, інтерпретувати наміри користувача та формувати відповідь відповідно до заданих правил, алгоритмів або моделей. Завдяки цьому діалогові системи можуть виконувати функції різного рівня складності: від простого інформування до прийняття рішень, що базуються на історії взаємодії, контексті та попередньому досвіді [3].

1.1.2 Сфери застосування та приклади використання

Якщо узагальнювати сфери застосування, можна побачити, що чат-боти поширені там, де існує потреба в оперативній взаємодії, автоматизації рутинних процесів або забезпеченні постійної доступності сервісу. Їх використовують в навчальних середовищах, соціальних платформах, корпоративних системах, технічній підтримці, ігрових серверах, ком'юніті-просторах, а також у будь-яких цифрових середовищах, де відбувається комунікація між користувачами. Щоб показати різноманітність таких рішень, варто розглянути два найбільш поширених прикладів застосування чат-ботів у двох найбільших сферах їх використання.

Перший приклад демонструє використання чат-ботів у сфері бізнесу. У процесі зростання компанії та бізнесу збільшується кількість запитів від клієнтів, замовлень та звернень. Людські ресурси мають межі, а очікування користувачів навпаки, постійно зростають. У такій ситуації чат-бот стає інструментом, який дозволяє автоматизувати відповіді на типові питання, оптимізувати процеси обробки замовлень чи питань від користувачів, скоротити навантаження на персонал та забезпечити цілодобову доступність сервісу. Завдяки інтеграції з внутрішніми системами, найпоширенішими якими є CRM, бот може не лише консультувати, але й виконувати конкретні операції, наприклад створювати замовлення, відповідати на типові питання або ж передавати складні запити операторам. Відтак, чат-бот підтримує масштабування бізнес-процесів і підвищує ефективність взаємодії з клієнтами без залучення додаткових ресурсів та коштів бізнесу. Ці переваги використання чат-ботів зображено на рисунку 1.1, де демонструється суттєве зниження середнього часу відповіді та збільшує пропускну здатність каналів комунікації порівняно з ручною обробкою.

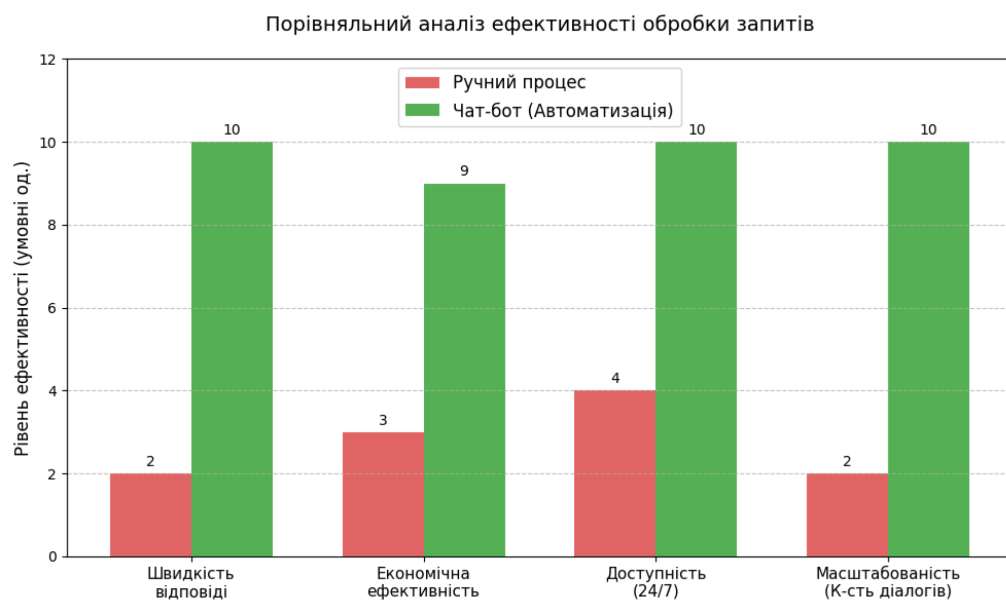

Рис 1.1. Графік порівняння ефективності обробки запитів

Що ж до другого прикладу, то він пов’язаний зі сферою цифрових спільнот та онлайн-ком’юніті. Чат-боти у таких середовищах виконують роль модераторів,

організаторів і навіть мають елементів геймплею(різні чат-ігри) або соціальної динаміки. На ком'юніті серверах у Discord, чи коментарях та спільних чатів в Telegram чи інших платформах вони автоматично фільтрують небажаний контент, видають ролі, ведуть логування активності, забезпечують навігацію по каналах, організовують події та опитування. У деяких випадках такі боти стають "ядром" роботи всієї спільноти, оскільки саме вони забезпечують порядок, доступність інформації й інтерактивність. Користувачам не потрібно вручну шукати потрібні ресурси або чекати реакції адміністратора – бот чергує постійно й відповідає миттєво. Отже, чат-бот в цій сфері виконує функцію координатора, полегшує управління великою кількістю учасників та формує структуроване середовище для комунікації.

Обидва приклади описують найважливіший параметр сучасного чат-бота – гнучкість. Чат-боти повинні адаптуватися під контекст використання, відтворюючи функціональність, яка відповідає потребам конкретного середовища. Саме ця властивість робить їх універсальним механізмом, що поєднує автоматизацію, доступність та інтерактивність в одній єдиній системі.

1.1.3 Узагальнена архітектура та компоненти чат-бота

Структура сучасного чат-бота, незалежно від сфери його застосування, базується на низці взаємопов'язаних компонентів, кожен з яких виконує окрему функцію в процесі обробки запиту користувача. Незважаючи на різноманіття платформ, протоколів та цілей використання, архітектурна логіка більшості діалогових систем залишається подібною, що дозволяє класифікувати загальні етапи взаємодії людини з ботом.

На рисунку 1.2 подано узагальнену архітектурну модель чат-бота, яка відображає ключові стадії обробки запиту: аналіз повідомлення, визначення наміру, управління діалогом, отримання або обробку даних і формування відповіді. Така модель демонструє, що будь-який чат-бот функціонує як цілісна система, у якій ланцюжок перетворень даних забезпечує перехід від "вхідного повідомлення" до "осмисленої відповіді".

Першим елементом архітектури виступає модуль розуміння мови(Language Understanding), який здійснює лінгвістичний та семантичний аналіз повідомлення. На цьому етапі система ідентифікує інтенцію користувача, визначає супровідний контекст і класифікує тип запиту. Цей компонент може будуватися як на правилах, так і на моделях машинного навчання, що дозволяє збільшувати точність інтерпретації з часом.

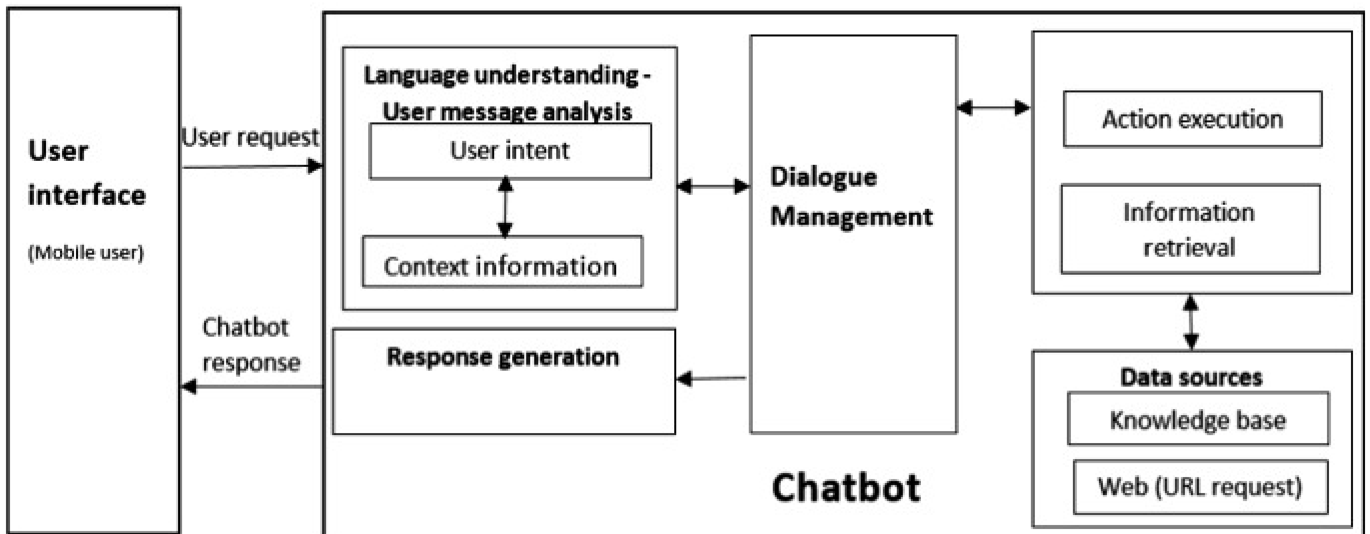

Рис. 1.2. Загальна архітектура чат-бота

Далі інформація надходить до модуля управління діалогом(Dialogue Management). Він відповідає за визначення подальших дій системи: виклик зовнішнього сервісу, звернення до бази знань, уточнення інформації або формування відповіді. Фактично, цей модуль виконує роль "логічного ядра" чат-бота, оскільки саме тут приймаються рішення на основі намірів користувача та попереднього перебігу діалогу. Для виконання дій чат-бот спирається на блок роботи з даними(Data Sources), який може включати внутрішні бази знань, зовнішні API, корпоративні системи або інші джерела інформації. Саме цей компонент надає доступ до фактів, операцій або структурованих відомостей, необхідних для побудови відповіді або виконання запиту.

На завершальному етапі працює модуль генерації відповіді(Response Generation), який формує результат у вигляді текстового повідомлення. Цей модуль

адаптовує інформацію у зрозумілий, логічно зв’язний та контекстно доречний формат. Такий модульний підхід дозволяє розробити центральну логіку бота для будь-якої сфери, яка може бути інтегрована з різними каналами зв'язку лише шляхом адаптації вхідного та вихідного модулів [4].

1.2 Аналіз архітектурних моделей взаємодії чат-ботів з платформами

1.2.1 Принципи та класифікація архітектурних моделей

Розвиток діалогових систем і зростання популярності чат-ботів у різних цифрових середовищах зумовили появу кількох стандартних архітектурних моделей, які визначають спосіб взаємодії між ботом і платформою. Вибір архітектури залежить не лише від технічних можливостей платформи, але й від вимог до швидкодії, стабільності, безпеки та масштабу системи. Незважаючи на спільну мету в вигляді забезпечення обміну повідомленнями між користувачем і ботом, ці архітектури суттєво відрізняються способом передачі даних, механізмами ініціації запитів та роллю сервера у процесі обробки взаємодій.

Формування архітектури чат-бота починається з визначення того, хто є ініціатором комунікації: чи сервер самостійно звертається до платформи, чи платформа передає запит безпосередньо до сервера, чи система працює як автономний сервіс всередині певної інфраструктури. Саме тому у практиці розробки найпоширенішими є три підходи:

1. модель опитування(Polling);
2. модель веб-хуків(Webhook);
3. модель сервісної інтеграції(Service-based Architecture).

Кожна з них відображає різні принципи організації трафіку та керування подіями, що, у свою чергу, впливає на гнучкість, швидкість реакції та архітектурну складність усього програмного рішення.

1.2.2 Модель періодичного опитування(Polling)

Першою з розглянутих моделей є архітектура, що базується на механізмі

періодичного опитування платформи. У цій моделі саме бот ініціює всі запити, регулярно звертаючись до сервера платформи із запитом на отримання нових подій. Такий режим можна порівняти з циклічним чергуванням, у якому бот безперервно перевіряє, чи не з'явилося нового повідомлення.

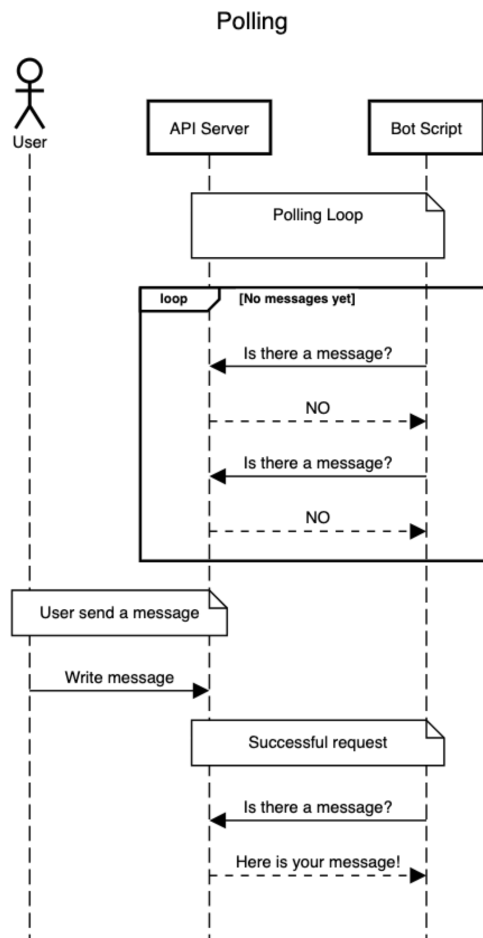

Рис. 1.3. Архітектурна модель взаємодії чат-бота на основі Polling

Попри відносну простоту реалізації, така схема має очевидні недоліки. Затримка між подією та реакцією прямо залежить від частоти запитів, а надмірно часте опитування збільшує навантаження на сервер і неефективно використовує ресурси. Водночас саме polling часто стає початковим рішенням для невеликих проєктів або систем, які не потребують обробки великих обсягів трафіку. Сильною стороною цього метода є мінімальна вимога до інфраструктури, адже сервер розробника може працювати в будь-яких умовах без спеціального зовнішнього доступу, що робить цю модель надзвичайно популярною в навчальних, тестових і

середніх за масштабом системах. На рисунку 1.3 зображено принцип функціонування цієї архітектурної моделі.

1.2.3 Подійно-орієнтована модель веб-хуків(Webhook)

Архітектура взаємодії чат-бота з платформою через механізм веб-хуків ґрунтується на подійно-орієнтованому принципі, де ініціатором передачі даних виступає не бот, а сервер платформи. Такий підхід забезпечує максимально швидкий обмін інформацією та мінімізує непотрібний мережевий трафік. Логіка роботи цієї моделі чітко структурована і відповідає етапам, які відображено на рисунку 1.4.

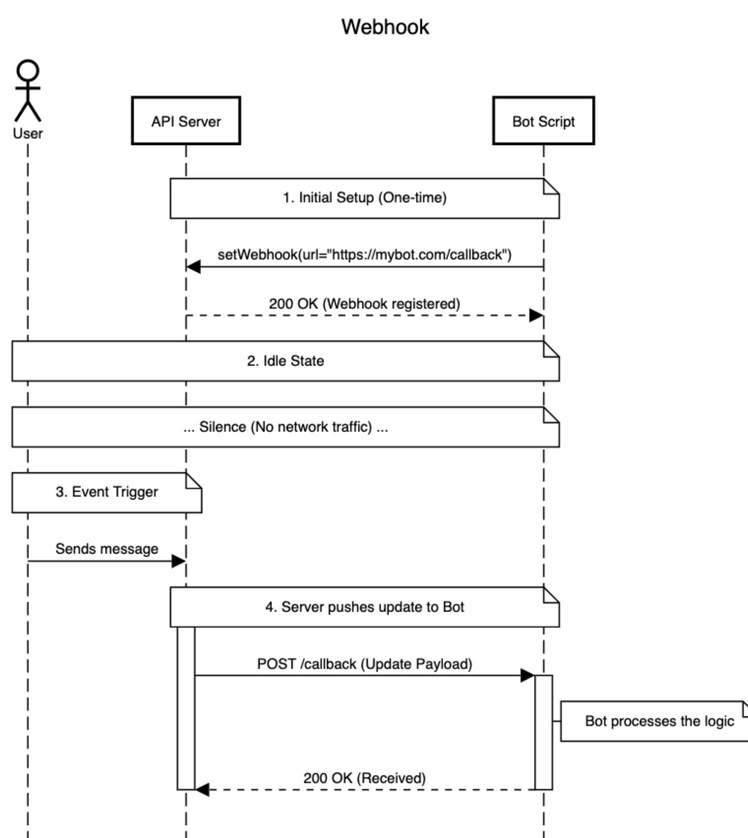

Рис. 1.4. Архітектурна модель взаємодії чат-бота на основі Webhook

Початковим кроком є одноразове налаштування, під час якого бот реєструє на сервері платформи адресу для зворотних викликів. Цей процес відбувається шляхом передачі спеціального запиту, у якому задається публічна URL-адреса, призначена для приймання подій. Після отримання цього запиту платформа підтверджує

успішну реєстрацію веб-хука, надсилаючи відповідь зі статусом успішної операції. Після цього система переходить у пасивний стан очікування, що не супроводжується жодним мережевим навантаженням, адже сервер не ініціює запити самостійно та не здійснює жодного періодичного опитування.

Всі подальші дії залежать від появи нової події на платформі. Коли користувач надсилає повідомлення або виконує іншу дію, яка потребує обробки ботом, сервер формує структурований пакет даних, що містить інформацію про подію. Цей пакет передається за раніше зареєстрованою URL-адресою за допомогою HTTP-запиту типу POST. Таким чином сервер самостійно "штовхає" оновлення до бота без будь-яких запитів з боку клієнтської логіки.

Після отримання повідомлення програмний модуль бота розпочинає обробку логіки відповідно до змісту події, аналізує отримані дані, визначає намір користувача та генерує відповідь. У цей момент важливим є підтвердження отримання події, яке надсилається у вигляді відповіді зі статусом успішної обробки. Це підтвердження дає змогу серверу платформи гарантувати коректність доставки оновлення та уникнути повторних передач.

1.2.4 Архітектура сервісної інтеграції(Service-based)

Третю модель становить архітектура сервісної взаємодії, яка передбачає включення чат-бота в більш широке середовище програмних сервісів. У цьому підході бот функціонує як частина комплексної інфраструктури, де платформи постачають лише канали передачі даних, а основні обчислення, логіка, інтеграції та зберігання інформації переносяться до окремих служб. Така архітектура властива корпоративним рішенням, які інтегруються з CRM-системами, внутрішніми каталогами, аналітичними платформами або іншими компонентами, розміщеними у хмарі чи приватних дата-центрах. При цьому взаємодія між компонентами зазвичай організовується через чітко визначені API-контракти, що спрощує розвиток системи та заміну окремих модулів без впливу на канали комунікації. Цю архітектуру детально зображено на рисунку 1.5.

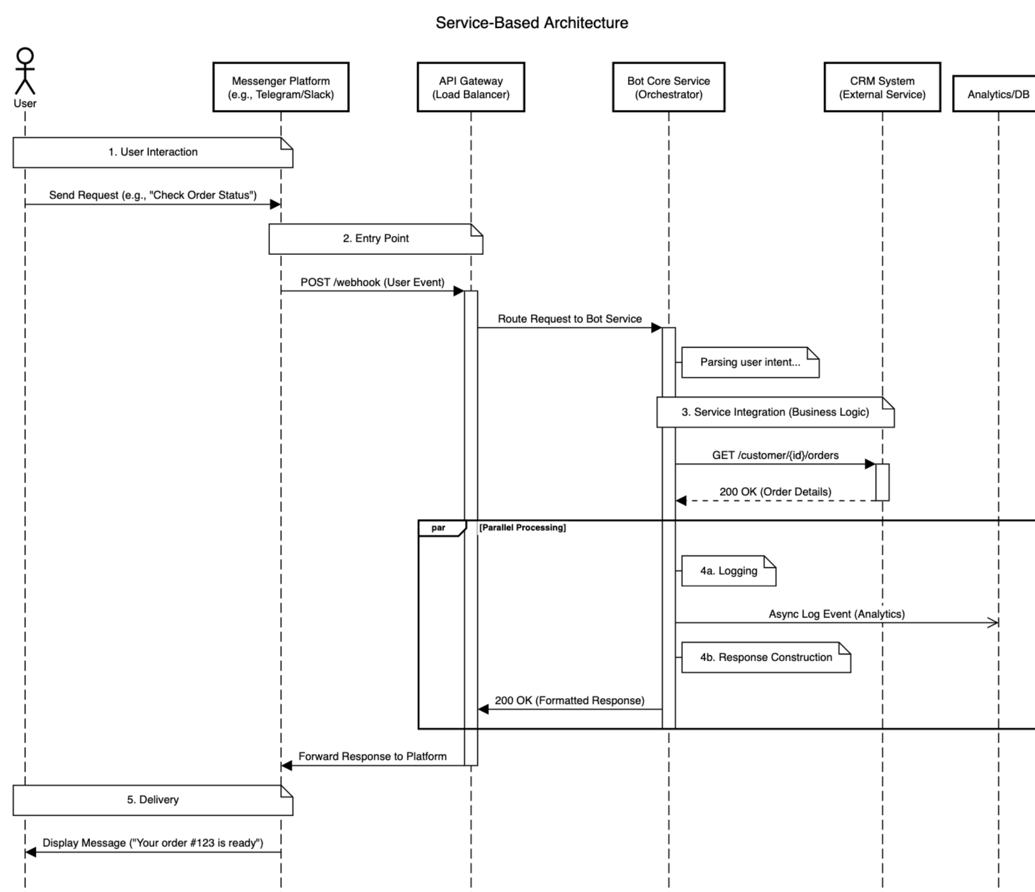

Рис 1.5. Архітектурна модель взаємодії чат-бота на основі Service-based

У системах цього типу канал комунікації між ботом і користувачем може реалізовуватися як через webhook, так і через інші механізми взаємодії, проте ключову роль відіграють мікросервіси, API-шлюзи та внутрішні компоненти логіки. Як наслідок, підхід service-based дозволяє масштабувати окремі частини системи автономно, підтримувати складну бізнес-логіку, легко додавати нові модулі та оптимізувати навантаження. Однак складність такої архітектури значно вища, і її впровадження потребує глибшого планування, контролю стабільності та системного моніторингу.

1.2.5 Порівняльний аналіз на основі інших досліджень

Для узагальнення відмінностей між цими архітектурними моделями нижче наведено порівняльну таблицю 1.1, яка демонструє основні характеристики кожного підходу.

Таблиця 1.1

Порівняльна характеристика моделей взаємодії чат-ботів з платформами

Параметр	Polling	Webhook	Service-based
Спосіб ініціації взаємодії	Бот самостійно опитує платформу	Платформа надсилає події боту	Взаємодія координується службами та API
Затримка реакції	Залежить від частоти опитувань	Мінімальна, подійна реакція	Залежить від внутрішньої інфраструктури
Вимоги до інфраструктури	Мінімальні	Потрібен сервер і SSL	Потребує складної інфраструктури
Масштабованість	Обмежена великим навантаженням	Висока за умов коректної конфігурації	Дуже висока, залежить від архітектури
Сфери застосування	Тестові, навчальні, невеликі проєкти	Продуктивні системи та комерційні боти	Корпоративні платформи та складні системи
Ресурсоефективні-сть	Низька при високому навантаженні	Висока, виклики лише за подією	Залежить від кількості сервісів
Гнучкість логіки	Обмежена простотою моделі	Гнучка, але прив’язана до платформ	Максимальна, логіка розподілена

Чат-боти і способи їх інтеграції з платформами слід розглядати не лише як технічну проблему передачі повідомлень, а як архітектурне та організаційне питання, що впливає на швидкодію, надійність і економічність системи. Як зазначають дослідники з Луцького національного технічного університету в своїй науковій роботі, вибір між моделями polling та webhook пов’язаний із низкою компромісів: простота реалізації й незалежність від зовнішніх ресурсів протистоять вимогам до швидкої реакції й безшовної масштабованості. демонструє практичні відмінності між long-polling та webhook на прикладі розміщення ботів у месенджері Telegram і описує, за яких умов кожен метод виправданий – від демонстраційних і

навчальних проєктів до промислових впроваджень, які потребують миттєвої обробки подій.

Натомість інший технічний огляд від професорів комп'ютерних технологій університету Айн-Шамс, який вони виклали в своїй науковій роботі, розглянули системну архітектуру чат-ботів, що деталізує внутрішні компоненти діалогових агентів і їхню взаємодію з зовнішніми сервісами, вказуючи на те, що сервісно-орієнтовані дизайни дають найбільшу гнучкість для складних, інтегрованих рішень, але вимагають ретельного планування, моніторингу й налаштування інфраструктури [5].

1.3 Використання штучного інтелекту для автоматизації діалогів

1.3.1 Принципи роботи чат-ботів без використання мовних моделей

У традиційних чат-ботах діалог будується на основі заздалегідь визначених правил, словникових шаблонів та формальних сценаріїв. Робота таких систем зводиться до аналізу вхідного повідомлення через механізм пошуку ключових слів або текстових патернів, після чого бот повертає наперед підготовлену відповідь. У випадках, де необхідно забезпечити більш логічну структуру взаємодії, використовуються станові машини, у межах яких кожен крок діалогу відповідає певному стану, а перехід до іншого стану залежить від того, яку фразу обрав користувач. Подібний підхід дає змогу відтворювати передбачувані й контрольовані сценарії, але водночас істотно обмежує гнучкість системи. Такі чат-боти фактично не розуміють смислового навантаження повідомлень, не аналізують контекст попередніх діалогів і не здатні адекватно реагувати на неточні, емоційні або неоднозначні висловлювання. Будь-яке розширення їх функціональних можливостей вимагає ручної модифікації сценаріїв, а зміни в логіці часто ведуть до необхідності повного перепроєктування структури діалогу чат-бота.

1.3.2 Переваги використання штучного інтелекту в чат-ботах

Поява великих мовних моделей відчутно змінила підхід до створення

чат-ботів. Якщо раніше можливості таких систем обмежувалися набором жорстких правил та сценаріїв, то сьогодні штучний інтелект дозволяє значно глибше розуміти зміст повідомлень і реагувати на них більш природно. Дослідження показують, що сучасні моделі добре працюють навіть тоді, коли користувачі роблять помилки, формулюють думки неповно або користуються неформальною лексикою. Це значно відрізняє їх від традиційних сценарних ботів, які часто "ламаються" на нестандартних фразах і потребують суворої відповідності сценарію.

Найбільш помітною перевагою є здатність штучного інтелекту підтримувати контекст розмови. Модель враховує попередні повідомлення, пам'ятає тему, може повернутися до деталей, згаданих раніше. Завдяки цьому діалог стає логічним і послідовним, а сам чат-бот сприймається як система, що розуміє користувача, а не просто реагує на окремі слова. Дослідники зазначають, що багато опитуваних компаній зазначають можливість ШІ до такого "природного спілкування" як один з головних факторів їх впровадження в свої чат-системи [6].

Рис. 1.6. Робочий процес NLP та контекстного ШІ

Не менш важливе значення також має розвинута персоналізація. На основі зібраних даних про поведінку користувача, історію його звернень, типові сценарії та попередні вибори, чат-бот з ШІ може адаптувати тон, зміст і рівень деталізації відповіді під конкретну людину. Практично це реалізується через механізми контекстної пам'яті та формування узагальненого профілю взаємодії, що дозволяє швидше відновлювати релевантний контекст діалогу і пропонувати більш доречні рекомендації. Водночас персоналізація має супроводжуватися дотриманням принципів конфіденційності та прозорими правилами використання даних, оскільки саме довіра користувача визначає прийнятність такого підходу. У наукових роботах, присвячених впливу AI-ботів на клієнтський досвід, персоналізація розглядається як

фактор підвищення довіри та лояльності: користувачі позитивно реагують на відчуття “індивідуального підходу” і більш охоче повертаються до таких сервісів [7].

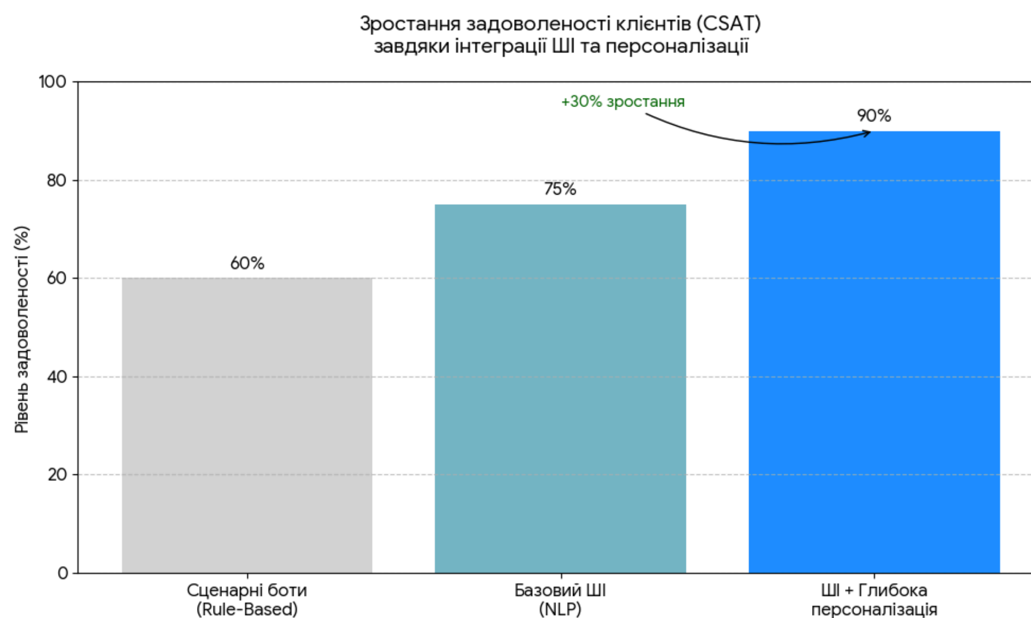

Рис. 1.7. Зростання задоволеності клієнтів завдяки впровадженню AI в чат-ботах

Як йдеться в дослідженні, в наш час інтеграція AI-компонентів є невід’ємною частиною архітектури, що наочно демонструється на рисунку 1.7, який ілюструє зростання задоволеності клієнтів(CSAT) завдяки впровадженню технологій штучного інтелекту та персоналізації у чат-ботах. За основу взяті дані з аналітичних звітів консалтингових компаній Gartner та McKinsey, які свідчать про те, що в наші дні перехід від простих сценарних ботів до інтелектуальних систем підвищив задоволеність клієнтів на 15-30% [8].

Також дослідники відмічають аналітичний потенціал розширення чат-ботів. В таких системах можна збирати та агрегувати дані про найпоширеніші питання, типові помилки як моделі, так і користувачів, зрозуміти найбільш проблемні місця в своїх системах, що дозволяє легко виявити проблемні процеси та своєчасно коригувати як логіку діалогу, так і самі послуги. У наукових роботах, присвячених оцінюванню LLM в чат-ботах, наголошується, що саме поєднання діалогового інтерфейсу з аналітикою на основі великих мовних моделей створює підґрунтя для безперервного вдосконалення системи та її адаптації до змін потреб користувачів [9].

1.3.3 Підходи до інтеграції штучного інтелекту у веб-системи

Інтеграція мовних моделей у веб-додатки найчастіше реалізується через звернення до зовнішніх API провайдерів ШІ. У цьому випадку веб-система або окремий модуль чат-бота приймає повідомлення від користувача, виконує попередню нормалізацію тексту, формує запит до сервісу штучного інтелекту та отримує у відповідь згенерований текст, який далі адаптується під формат цільового інтерфейсу. Така схема дає змогу відносно швидко додати інтелектуальний діалоговий інтерфейс навіть до вже наявних систем, не розгортаючи власну ML-інфраструктуру, і широко описується як у прикладних гайдах, так і в наукових роботах, присвячених практичним аспектам використання LLM в чат-ботах.

Однак у багатьох організаціях питання конфіденційності та контролю над даними стоять на першому місці. У цьому підході мовна модель працює в межах власної інфраструктури – на внутрішніх серверах чи у приватній хмарі. Такий варіант дає змогу уникнути передачі чутливої інформації третім сторонам та гнучко налаштовувати режим роботи системи відповідно до корпоративних політик безпеки. Дослідження, присвячені впровадженню LLM в чат-ботах у чутливих доменах(освіта, медицина, фінансовий сектор), описують, що саме локальний або гібридний підхід дозволяє поєднати переваги сучасних моделей із суворими вимогами до захисту даних.

У більш комплексних системах інтеграція ШІ реалізовується через мікросервісну архітектуру. Модель у такому випадку працює як окремий сервіс, з яким взаємодіють інші компоненти – модуль управління контекстом, система модерації відповідей, маршрутизатор запитів, аналітичний модуль. Така архітектура при збільшенні навантаження досить легко розширюється, достатньо виділити моделі більше ресурсів або додати нові екземпляри сервісу. Крім того, поділ на мікросервіси покращує керованість і значно спрощує оновлення та розширення функціональності [10].

Удосконалення діалогових систем вимагає переходу до гібридних архітектур, здатних ефективно поєднувати моделі штучного інтелекту з логікою управління розподіленим середовищем. Такий підхід передбачає, що за змістовне та природне

генерування відповідей відповідає LLM, тоді як системні функції, такі як збереження історії діалогу, оновлення стану та маршрутизація запитів винесені на окремі модулі. Ключовою проблемою, яку вирішує гібридний метод, є передача контексту розмови до LLM у розподіленому середовищі, що досягається шляхом формування динамічного промпта на основі поточного стану діалогу та зовнішніх джерел даних. Рисунок 1.8 наочно демонструє цей інтеграційний процес: від отримання вхідного повідомлення та вилучення необхідного контексту на формування промпта, обробки його LLM і повернення осмисленої, контекстно-залежної відповіді користувачу [11].

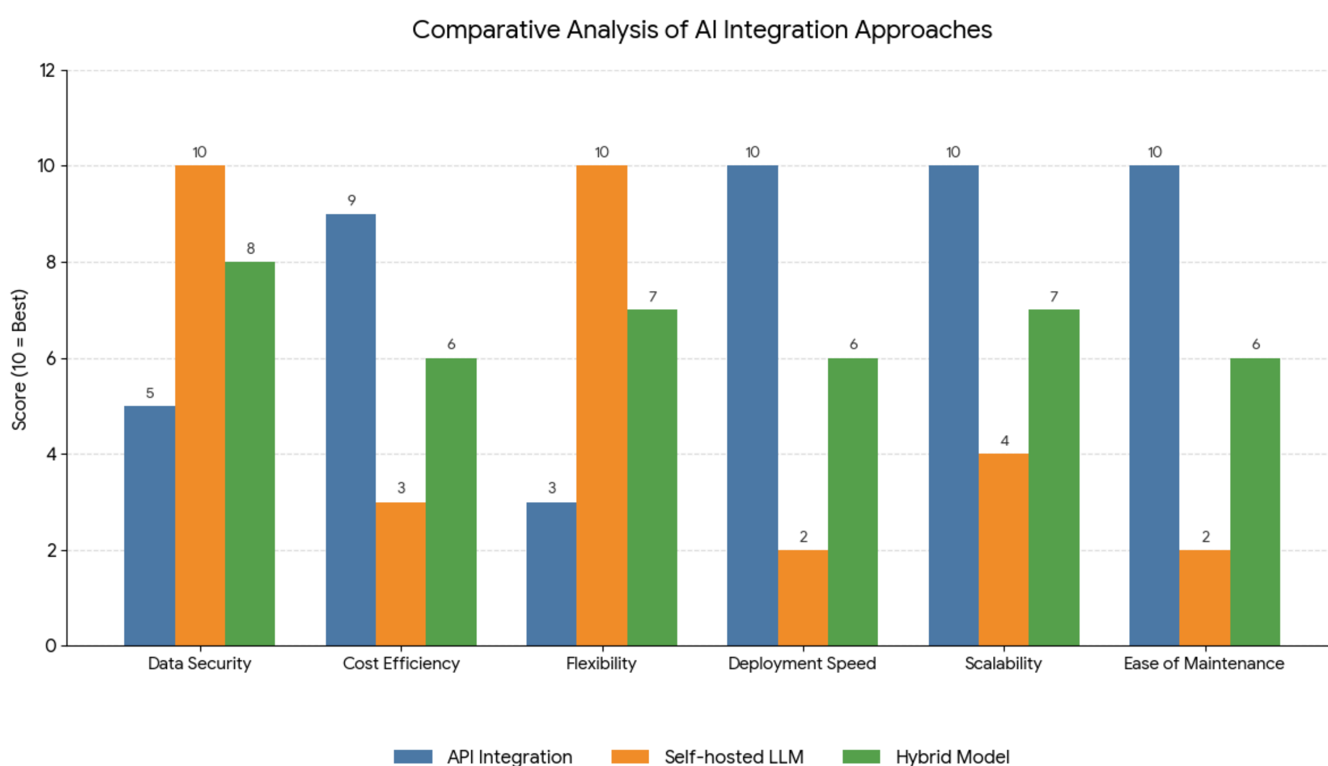

Рис. 1.8. Порівняльний аналіз архітектурних підходів до інтеграції ШІ

Вибір архітектурного підходу для інтеграції ШІ у корпоративні системи є стратегічним рішенням, що базується на балансі між вартістю, безпекою та гнучкістю. Як ілюструє аналіз на рисунку 1.8, кожен із методів має свої переваги та недоліки, підтверджені сучасними дослідженнями. API-інтеграція є найбільш поширеним методом завдяки своїй простоті та низькому порогу входження. Згідно зі звітом Databricks, використання SaaS LLM(наприклад, ті ж OpenAI) зросло на 1310%

за рік, що підтверджує його лідерство у швидкості впровадження. Однак цей підхід має суттєві ризики щодо безпеки даних, оскільки чутлива інформація залишає “периметр” компанії. Дослідження Andreessen Horowitz вказують на те, що хоча відкриті моделі дозволяють уникнути прив'язки до вендора, вони вимагають значних початкових інвестицій у інфраструктуру та мають високу складність підтримки [10-12].

Вищезгадана гібридна модель дозволяє поєднати переваги обох підходів: використовувати потужні хмарні API для загальних задач та локальні моделі для обробки конфіденційних даних. Це забезпечує оптимальний баланс гнучкості та безпеки, що підтверджується трендами переходу підприємств до мультимодельних архітектур [13].

Окремий вимір інтеграції – це користувацький досвід в її використанні. Навіть якісно налаштована модель може сприйматися користувачами як “повільна” або “незручна”, якщо чат-інтерфейс інтегровано невдало: відповіді надходять із затримкою, інтерфейс перевантажений елементами або, навпаки, не дає зрозуміти, які дії доступні. Систематичні огляди, присвячені AI-чатботам у контексті клієнтського досвіду, показують, що користувачі схильні позитивно оцінювати системи, де поєднані швидкість відповіді, зрозуміла структура діалогу та відчутна персоналізація [12-13].

Загалом інтеграція штучного інтелекту не обмежується технічним підключенням моделі. Це багатошаровий процес, який включає вибір архітектури, забезпечення безпеки, адаптацію інтерфейсу та формування внутрішніх правил обробки даних. Урахування цих нюансів дозволяє створювати надійні системи, здатні працювати з великими обсягами запитів і підтримувати високу якість діалогу.

Висновок до розділу

У цьому розділі проведено теоретичний аналіз технологій та методів побудови і інтеграції чат-ботів, а також розглянуто архітектурні моделі взаємодії, такі як Polling, Webhook, Service-based та підходи до автоматизації діалогів.

Дослідження показало, що існуючі підходи мають низку недоліків, які впливають на всю схему. Зокрема, реалізація чат-ботів у різних платформах суттєво відрізняється через фрагментацію технологій, оскільки можуть використовуватися власні API чи специфічні формати даних, що призводить до обмеженої гнучкості, несумісності рішень та ускладнює їх централізоване управління. Водночас, використання традиційних сценарних ботів не здатне задовольнити сучасні вимоги до “розумної” поведінки, підтримки контексту та персоналізації.

Було визначено, що відсутність уніфікованих моделей для нормалізації даних та складність інтеграції великих мовних моделей(LLM) у багатоплатформне середовище є основним бар'єром для створення комплексних та масштабованих діалогових систем.

Це підтверджує актуальність розробки уніфікованих моделей, методів та алгоритмів, які дозволяють абстрагуватися від технічних особливостей конкретних месенджерів та інтегрувати різнорідні чат-боти у єдину екосистему на основі моделі програмного забезпечення як послуги. На основі порівняння моделей взаємодії та їх обмежень було сформовано необхідні критерії до інтеграційного рішення: масштабованість, уніфікація форматів повідомлень і централізована керованість. Зазначені критерії визначають очікувані переваги майбутньої системи, такі як спрощення підключення нових месенджерів, зниження фрагментації даних та створення передумов для стабільнішого збереження контексту і підвищення якості діалогів.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ АРХІТЕКТУРНИХ ПІДХОДІВ ТА МЕТОДІВ ПРОЄКТУВАННЯ СИСТЕМ ДЛЯ ЧАТ-БОТІВ

2.1 Методика побудови багатоплатформенного інтеграційного шару

2.1.1 Вимоги до інтеграційного шару в багатоплатформеному середовищі

Як показав аналіз архітектурних моделей у попередньому розділі, найбільшою проблемою інтеграції чат-ботів є фрагментація технологій: кожен месенджер має власний API, формат подій та вимоги до транспортного рівня, що ускладнює побудову єдиного програмного рішення для роботи з кількома каналами одночасно. У більшості промислових реалізацій інтеграційний шар формується еволюційно – як набір специфічних адаптерів, тісно пов’язаних із конкретними ботами, базами даних та бізнес-логікою. Такий підхід породжує сильну взаємозалежність компонентів системи, оскільки збій в одному модулі може блокувати роботу всієї системи, а внесення змін до логіки обробки подій вимагає повного перезапуску, зазвичай монолітного, застосунку [14].

З позицій дослідження інтеграційних рішень для діалогових систем інтеграційний шар доцільно розглядати як окремий, відносно автономний рівень, що виконує функції нормалізації повідомлень, маршрутизації, агрегації подій і взаємодії із зовнішніми сервісами, зокрема моделями штучного інтелекту. У такому формулюванні до нього висуваються як функціональні, так і нефункціональні вимоги. До функціональних належать підтримка кількох платформ(Telegram, Viber, WhatsApp тощо), єдина модель повідомлення, можливість підключення різних ботів і сценаріїв без зміни ядра системи. Нефункціональні вимоги охоплюють масштабованість, відмовостійкість, спостережуваність та можливість поетапного оновлення окремих компонентів без простою сервісу [15].

Проблема традиційних архітектур полягає у тісному поєднанні транспортного рівня, діалогової логіки та інтеграцій з бекенд-сервісами в одному монолітному процесі, що значно ускладнює підтримку та додавання нових месенджерів. У таких

системах будь-яка зміна в протоколі конкретної платформи спричиняє необхідність перегляду взаємодії між усіма компонентами, що підвищує ризик регресій та ускладнює тестування. Це, у свою чергу, призводить до зниження прозорості логіки обробки подій та зростання ймовірності помилок, пов'язаних із неврахованими відмінностями між API різних платформ [14-16].

Окремою групою проблем є відсутність уніфікованої моделі подій, оскільки один і той самий тип взаємодії користувача може бути поданий у Telegram, Viber або інших веб-чатах різними за структурою об'єктами, що відрізняються набором атрибутів, вкладеністю та способом ідентифікації джерела. Зі збільшенням кількості платформ така різнорідність призводить до експоненційного зростання складності обробки – кожен новий канал збільшує кількість можливих комбінацій форматів подій, що ускладнює забезпечення їх узгодженості. Ці проблематики демонструє рисунок 2.1, на якому всі проблематики виділені в "Проблеми інтеграційного шару".

Рис. 2.1. Основні проблеми багатоплатформенного інтеграційного шару

Також спостерігається проблема непослідовності та асинхронності надходження подій. Різні платформи використовують різні моделі доставки, що були згадані в попередньому розділі. Через це повідомлення можуть надходити у неправильному порядку, із затримками або з частковою втратою контексту. Така

поведінка ускладнює відтворення повної картини взаємодії користувача із системою, що негативно позначається на якості діалогу та точності подальшої обробки [17].

Узагальнюючі праці з архітектури чат-ботів підкреслюють, що найвразливішим місцем традиційних систем є саме тісне поєднання транспортного рівня, діалогової логіки та інтеграцій з бекенд-сервісами в одному процесі. У випадку багатоплатформенних рішень це означає, що кожен новий канал або бот "вростає" у моноліт, посилюючи залежності між модулями. Отже, методика побудови інтеграційного шару в рамках даного дослідження має спиратися на принципи модульності та слабкого зв'язку між компонентами, що природно приводить до розгляду мікросервісної архітектури та контейнеризації.

2.1.2 Проблеми мікросервісної організації інтеграційного шару

Розглядаючи інтеграційний шар як окремий рівень системи, логічно застосувати до нього мікросервісний підхід. У межах даного дослідження мікросервісом вважається автономний сервіс із чіткою зоною відповідальності, власним інтерфейсом взаємодії та можливістю незалежного розгортання. Для інтеграції чат-ботів це означає поділ монолітної логіки на окремі сервіси: приймання подій від платформ, нормалізацію повідомлень, маршрутизацію, керування контекстом діалогу, виклик моделей штучного інтелекту, логування й аналітику [18].

З одного боку, така декомпозиція безпосередньо відповідає вимогам, оскільки мікросервісний інтеграційний шар дозволяє послабити зв'язок між доменами. Сервіс, що обробляє події месенджера Telegram, більше не "зшитий" жорстко з логікою чату чи внутрішнього корпоративного месенджера. Іншими словами: зміна одного сценарію не вимагає перекомпіляції та перезапуску всієї системи. Крім того, мікросервіси створюють природне середовище для дослідження – легко виділити окремий сервіс, змінити його реалізацію і оцінити вплив на час відповіді, стійкість або використання ресурсів [19].

Результатом аналізу та експериментальних спостережень свідчать, що мікросервісний підхід, попри очевидні переваги, формує й власний спектр проблем. Однією з найважливіших є те, що система набуває розподіленого характеру, бо

обробка навіть одного користувацького повідомлення може включати низку послідовних викликів між різними сервісами. Кожен такий виклик додає додаткову затримку та створює потенційну точку відмови. Якщо для монолітних рішень більшість помилок локалізується в межах одного процесу, то у розподіленій системі збої окремих компонентів здатні породжувати «ланцюгові реакції» у вигляді накопичення черг, тайм-аутів і повторних спроб доставки [20].

Ще однією суттєвою проблемою стає забезпечення узгодженості даних. Оскільки інформація про діалог розміщується у кількох джерелах – частково у сховищі контексту, частково у журналах подій, частково у кешах – виникає потреба гарантувати синхронізацію станів між сервісами. У певних ситуаціях повідомлення вже може бути опрацьоване одним компонентом, але залишатися невідомим іншому, що призводить до розбіжностей між фактичним перебігом діалогу та тим, який сприймає користувач [21].

Додаткові складності виникають у сфері спостережуваності та моніторингу. Для аналізу роботи системи або діагностики затримок у відповіді необхідно простежити повний маршрут повідомлення, який проходить через декілька сервісів. За відсутності централізованих механізмів логування, трасування запитів і збору продуктивних метрик мікросервісна платформа фактично перетворюється на «чорну скриньку», підтримка якої виявляється значно складнішою порівняно з добре структурованим монолітом [22].

На рисунку 2.2 схематично зображено мікросервісну організацію інтеграційного шару, де окремі сервіси виконують автономні функції, а червоними стрілками позначено потенційні “вузькі місця”, у яких можуть накопичуватися затримки. Тут зображено шлях від сервісу нормалізації до сервісу маршрутизації та далі до сервісу взаємодії з AI і сервісу логування характеризується послідовністю синхронних викликів: кожен наступний сервіс очікує завершення попереднього, що в умовах навантаження збільшує загальний час відповіді бота.

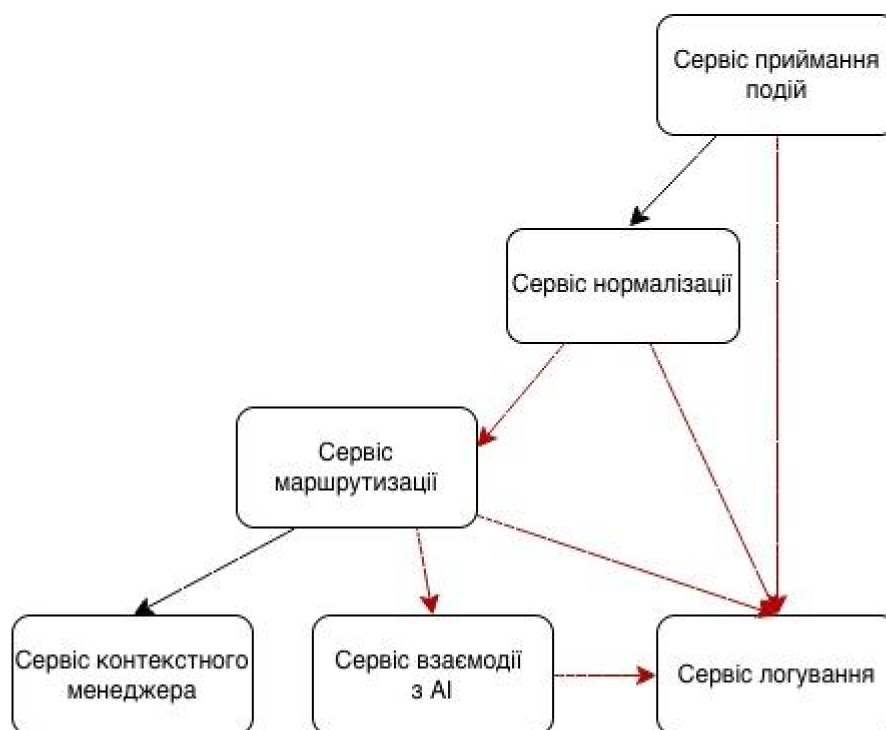

Рис 2.2. Мікросервісна організація інтеграційного шару

Додатково, звернення до AI-компонента є обчислювально й мереживо найдорожчою операцією, тому саме там затримка зростає найбільше. Окремо виділено червоними лініями потоки до сервісу логування, оскільки під час активного логування операції запису можуть бути повільнішими за обробку самих подій і ставати джерелом блокувань для інших сервісів.

2.1.3 Самостійні чат-боти як автономні виконавчі модулі

У платформенних системах чат-ботів окремий бот зазвичай реалізується як самостійний виконавчий модуль, що обслуговує одну платформу або один бізнес-сценарій. Такий бот має власну логіку обробки подій, власні налаштування підключення до API месенджера та окремий життєвий цикл. З огляду інтеграційного шару він сприймається як незалежний учасник обміну повідомленнями, до якого надходять нормалізовані події і від якого очікуються результати у уніфікованому форматі. Використання самостійних ботів створює зручні умови для поділу відповідальності та ізоляції логіки, оскільки кожен модуль відповідає за власний сценарій або платформу[23].

Разом з тим дослідження показує, що подібна організація породжує низку системних проблем, які ускладнюють підтримання цілісності всієї багатоплатформенної системи. Однією з ключових є дублювання логіки: перевірки прав доступу, робота з контекстом, формування відповідей і обробка часто повторюваних подій реалізуються у кожному боті окремо. У результаті зміна загальних правил потребує оновлення декількох модулів, що збільшує ризик неузгодженості та розходжень у поведінці ботів на різних платформах.

Разом з тим дослідження показує, що самостійні боти ускладнюють забезпечення цілісності системи. По-перше, за відсутності єдиного інтеграційного шару логіка обробки подій частково дублюється в кожному боті: перевірки прав доступу, однакові шаблони відповідей, загальні правила роботи з контекстом. Це призводить до розходжень у поведінці ботів на різних платформах та утруднює внесення глобальних змін. По-друге, зростає кількість точок адміністрування: необхідно окремо відстежувати стан кожного бота, контролювати версії та конфігурації, слідкувати за логами й метриками[24-29].

Ще однією проблемою є зростання кількості точок адміністрування. Оскільки кожен бот є самостійним процесом, необхідно окремо контролювати його конфігурацію, версію, стан підключення до платформи, журнали подій та метрики продуктивності. Це ускладнює забезпечення стабільності всієї системи, підвищує навантаження на команду супроводу та створює додаткові ризики, пов'язані з асинхронними збоями в окремих компонентах.

Суттєвої уваги потребує і питання цілісності аналітики. Події, що стосуються діалогів, взаємодій із контекстом та реакцією користувачів, розподіляються між кількома автономними ботами. За таких умов дані мають різний рівень деталізації, різні формати та різні часові мітки. Це ускладнює побудову єдиної картини роботи системи, перешкоджає своєчасному виявленню аномалій та унеможливлює повноцінне порівняння ефективності окремих ботів між собою. Фрагментація аналітичної інформації створює ситуації, коли загальні закономірності залишаються невидимими, а локальні проблеми помилково сприймаються як зовнішні або випадкові [30].

На рисунку 2.3 наведено залежність умовного індексу складності інтеграційної підсистеми від кількості автономних чат-ботів. Для побудови графіка використано модель, у якій дублювання логіки зростає лінійно з кількістю ботів, а кількість потенційних точок відмови оцінюється пропорційно числу парних взаємодій між модулями. Сукупний індекс складності $C(n)$ обчислювався як сума цих двох складових, що приводить до квадратичного характеру зростання.

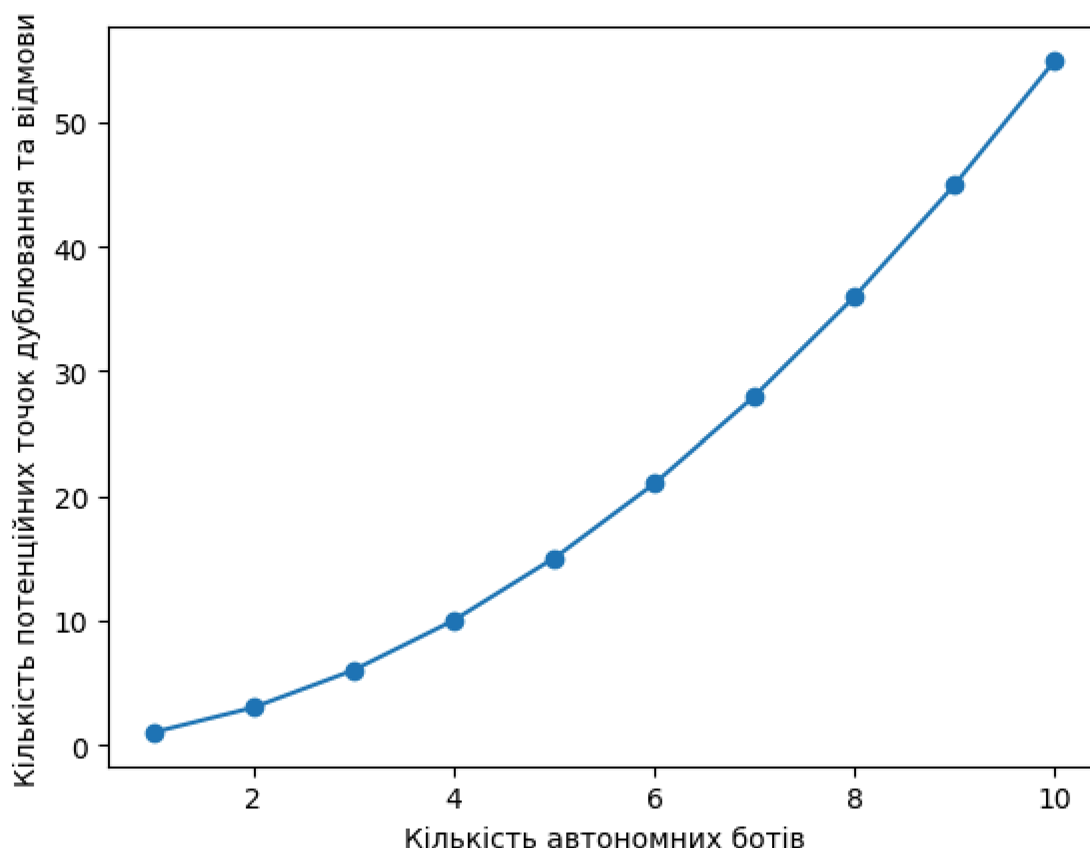

Рис 2.3. Графік зростання складності в впровадженні автономних ботів

Видно, що навіть за відносно невеликої кількості ботів показник складності зростає прискореними темпами, що підкреслює негативний вплив фрагментації та дублювання функцій на керованість системи. Разом всі ці факти свідчать про те, що використання самостійних чат-ботів як окремих виконавчих модулів створює не лише гнучкість у розробці, але й суттєві труднощі в управлінні, моніторингу та узгодженості поведінки системи в цілому.

2.2 Дослідження архітектури бази даних для зберігання повідомлень чат-ботів

2.2.1 Проблематика збереження повідомлень у базах даних

У більшості сучасних месенджерів механізм взаємодії з ботами побудований на основі подійної моделі, відомої як commit message. Платформа не надає повної історії діалогу, а надсилає окремі інкрементальні події, серед яких можуть бути нові повідомлення, реакції, редагування, видалення та інші зміни стану. Це означає, що система не володіє завершеним "знімком" розмови, а повинна самостійно збирати його з потоку подій, які надходять у різний час і з різною повнотою. Як наслідок, розробка простої універсальної структури даних для подібних повідомлень виявляється неможливою, оскільки події різних платформ суттєво відрізняються за складом, вкладеністю, обсягом метаданих та форматом передачі. Ця проблема зображена на рисунку 2 [31].

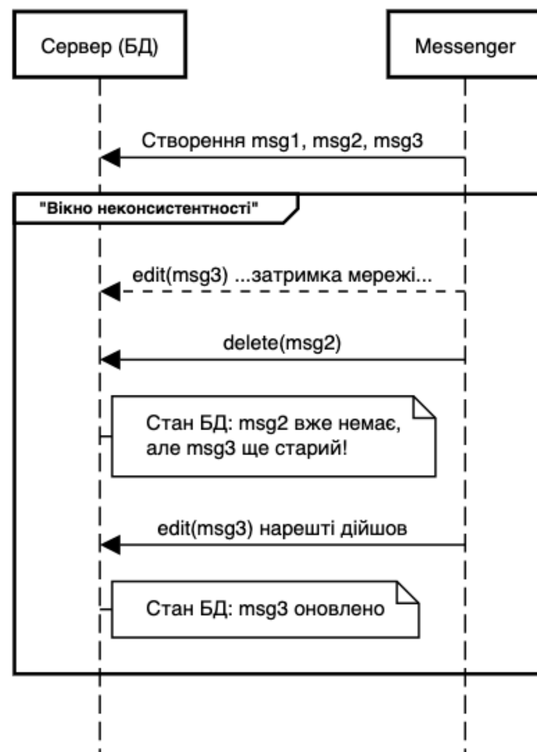

Рис. 2.4. Схема розбіжності станів повідомлень з потоку подій

Проблема ускладнюється тим, що зберігати повідомлення з різних платформ у окремих таблицях також виявляється недоцільним. Така фрагментація схеми бази

даних призводить до втрати узгодженості, ускладнює агрегування статистики, збільшує кількість механізмів підтримки та створює ситуації, коли однакова логіка має дублюватися у декількох місцях. Подібне розділення не спрощує систему, а лише робить її більш неоднорідною, знижує прозорість поведінки та ускладнює проведення міжплатформенного аналізу [32].

Суттєві труднощі виникають і у зв'язку з тим, що commit-події надходять не в упорядкованому вигляді. Через мережеві затримки, тимчасову недоступність інтеграційного шару або помилки парсингу частина подій може надійти пізніше, потрапити в інші часові інтервали або навіть загубитися. У такій ситуації фактичний стан діалогу в месенджері та його локальне відображення в базі даних розходяться, і встановити момент, коли це сталося, часто неможливо. Додаткові проблеми створюють події редагування і видалення. Якщо зберігати лише кінцевий стан повідомлення, система втрачає інформацію про проміжні версії, що унеможливлює аудит і аналіз дій користувача. Якщо ж зберігати усі варіанти, розмір таблиці починає рости в геометричній прогресії. Окремо стоїть питання зникаючих повідомлень: деякі платформи видаляють контент назавжди, що робить повторне отримання неможливим, а отже система повинна або копіювати ці дані з ризиком порушення політик конфіденційності, або визнавати прогалини в історії як неминучість [33].

Таким чином, проблематика збереження повідомлень у багатоплатформеній системі полягає не лише у відмінностях форматів, а насамперед у подійній природі даних, відсутності повної історії, можливості розсинхронізації та неможливості створити просту універсальну модель, яка б одночасно забезпечувала узгодженість, зберігала первинну семантику подій і масштабувалася разом зі зростанням обсягів трафіку.

2.2.2 Неможливість застосування нереляційних сховищ як універсального вирішення

Попри те, що формат повідомлень різних платформ є суттєво відмінним і постійно змінюваним, використання нереляційних сховищ даних не усуває

фундаментальних обмежень, пов'язаних із commit-моделлю доставки подій. З першого погляду такі системи видаються хорошим вибором, оскільки вони допускають довільну структуру документа, не вимагають фіксованої схеми та здатні зберігати вкладені об'єкти у первинному вигляді. Саме тому часто виникає припущення, що документно-орієнтовані БД можуть забезпечити універсальний механізм зберігання повідомлень від різних платформ.

Однак у багатоплатформенних чат-ботах проблема полягає не у складності структури окремого повідомлення, а у непередбачуваності та нестабільності послідовності подій, що надходять у систему. Commit-модель передбачає передавання лише змін, а не повної історії, тому будь-які редагування, видалення або запізнілі події можуть порушити цілісність діалогу. Навіть якщо документ можна зберегти у довільному вигляді, система все одно не має гарантій, що отримала всі необхідні події, отримала їх у правильному порядку або що ці події відображають повний стан розмови. У такій ситуації структура сховища не вирішує першопричину проблеми [34].

Платформи можуть змінювати моделі подій без попередження, додавати нові поля, вилучати старі або змінювати їхні значення залежно від контексту взаємодії користувача. У документно-орієнтованій БД це неминуче призводить до накопичення об'єктів різних версій, що, у свою чергу, ускладнює аналіз, порівняння даних та реконструкцію історії взаємодії. Читання таких записів потребує складної логіки інтерпретації, яка повинна враховувати різні варіанти структури, і з часом ця логіка стає не менш громіздкою, ніж у реляційних системах.

Важливо підкреслити, що модель eventual consistency, притаманна багатьом нереляційним сховищам, також суперечить вимогам до відтворення історії подій. Якщо система гарантує узгодженість лише у підсумку, але не в момент отримання кожної окремої події, відновити точний хід діалогу після збою або затримки стає неможливо. Для багатьох сценаріїв чат-ботів це означає втрату критично важливої інформації.

Таким чином, нереляційні системи не ліквідують ключові труднощі моделювання діалогу, а лише зміщують їх з рівня структури БД на рівень логіки

обробки. Проблеми, пов’язані з неповнотою даних, нестабільністю порядку подій, появою різних версій структур та ускладненим аналізом, залишаються невирішеними. Це свідчить про те, що фундаментальним обмеженням є не спосіб зберігання запису, а подійна природа даних, які надходять у систему.

2.2.3 Оцінка часу повного зчитування таблиці повідомлень через API

У попередніх підрозділах було показано, що подійна модель формування історії діалогу створює суттєві труднощі для зберігання та подальшої обробки повідомлень. Одним із практичних наслідків цієї проблематики є зростання обсягів таблиці повідомлень та необхідність роботи з великими масивами даних у режимі посторінкового зчитування. Щоб оцінити, наскільки масштабування бази даних впливає на реальну швидкість доступу до інформації, доцільно виконати розрахунки часу, потрібного для повного отримання історії діалогу через API. Такі оцінки не розв’язують проблему структурування самих даних, але дозволяють визначити межі продуктивності системи, оцінити поведінку при максимальному навантаженні та зрозуміти, за яких умов доступ до історичних записів стає важливим фактором для роботи всієї платформи.

Нехай таблиця повідомлень містить N записів, а доступ до неї ззовні здійснюється від API через протокол HTTP з пагінацією. Припустимо, що за один запит API повертає не більше ніж k повідомлень, тоді кількість необхідних запитів можна знайти за наступною формулою:

$$R = \frac{N}{k} \tag{2.1}$$

Час обробки одного запиту складатиметься з двох основних компонентів: мережевої затримки та службових дій протоколу HTTP, що будуть представлені як L, та часу передавання даних, який залежить від розміру відповіді та пропускної здатності каналу B байт/с.

Якщо позначити середній обсяг повідомлення у відповіді API я S_{api} байт, то обсяг даних у одній сторінці становитиме $k \times S_{api}$, а час передавання можна представити в вигляді наступної формули:

$$T = \frac{k \times S_{api}}{B} \quad (2.2)$$

Тоді орієнтовний час повного послідовного зчитування всієї таблиці через HTTP можна обрахувати за наступною формулою:

$$T_{total} \approx R \times (L + T) = \frac{N}{k} \times (\frac{k \times S_{api}}{B}) \quad (2.3)$$

Для подальшого аналізу зафіксуємо характерні значення параметрів, узгоджені з середніми значеннями з аналогічних розрахунків:

1. $N = N_{max} \approx 7 \times 10^7$записів;
2. середній обсяг повідомлень у відповіді API $S_{api} \approx 2500$ байт;
3. пропускна здатність каналу між інтеграційним шаром та клієнтом – $B = 50$ Мбіт$/c \approx 6,25 \times 10^6$ байт/с;
4. ефективна затримка на один HTTP-запит з урахуванням особливості мережі та обробки $L = 0,2$ с.

Підставимо ці значення для кількох варіантів розміру сторінки k.

Варіант 1. Значення k дорівнює 50 повідомлень на сторінку. Розрахунок значень виглядає наступним чином:

$$R = \frac{7 \times 10^7}{50} = 1\,400\,000 \text{ запитів},$$

$$T = \frac{50 \times 2500}{6,25 \times 10^6} = \frac{125000}{6,25 \times 10^6} \approx 0,02 \text{ с},$$

$$T_{total} \approx 1\,400\,000 \times (0,2 + 0,02) = 1400000 \times 0,22 \approx 308000\, c \approx 85,56 \text{ год.}$$

Варіант 2. Значення k дорівнює 100 повідомлень на сторінку. Розрахунок значень виглядає наступним чином:

$$R = \frac{7\times10^7}{100} = 697\,000 \text{ запитів},$$

$$T = \frac{100\times2500}{6{,}25\times10^6} = \frac{250000}{6{,}25\times10^6} \approx 0,04 \text{ с},$$

$$T_{total} \approx 697\,000 \times (0,2 + 0,04) = 697000 \times 0,24 \approx 167280\, c \approx 46,5 \text{ год.}$$

Варіант 3. Значення k дорівнює 500 повідомлень на сторінку. Розрахунок значень виглядає наступним чином:

$$R = \frac{7\times10^7}{500} = 140\,000 \text{ запитів},$$

$$T = \frac{500\times2500}{6{,}25\times10^6} = \frac{1250000}{6{,}25\times10^6} \approx 0,2 \text{ с},$$

$$T_{total} \approx 140\,000 \times (0,2 + 0,2) = 140000 \times 0,4 \approx 56\,000\, c \approx 15,55 \text{ год.}$$

Варіант 4. Значення k дорівнює 1000 повідомлень на сторінку. Розрахунок значень виглядає наступним чином:

$$R = \frac{7\times10^7}{1000} = 70\,000 \text{ запитів},$$

$$T = \frac{1000\times2500}{6{,}25\times10^6} = \frac{2\,500\,000}{6{,}25\times10^6} \approx 0,4 \text{ с},$$

$$T_{total} \approx 70000 \times (0,2 + 0,4) = 70000 \times 0,6 \approx 42\,000\, c \approx 11,67 \text{ год.}$$

Для наочності результати проміжних розрахунків доцільно подати у вигляді порівняльної таблиці. Узагальнені значення дають змогу оцінити масштаби затримок при роботі з великими обсягами повідомлень та показують, що навіть незначна зміна параметрів посторінкового доступу може суттєво позначитися на продуктивності системи. Зведемо отримані значення до порівняльної таблиці 2.2.

Таблиця 2.2

Оцінка часу повного зчитування таблиці повідомлень через HTTP

кількість k повідомлень	Кількість запитів R	Час передавання одного T, с	Приблизний час T_{total} год
50	1 400 000	0,02	85,56
100	697 000	0,04	46,5
500	140 000	0,2	15,55
1000	70 000	0,4	11,67

З таблиці видно, що збільшення розміру повідомлень істотно зменшує сумарний час зчитування за рахунок зменшення кількості запитів HTTP, навіть попри зростання часу передавання кожної окремої сторінки. Іншими словами, при умові помітної затримки L домінують саме накладні витрати на велику кількість мережевих звернень, а не обсяг переданих даних. Для наглядної ілюстрації цих даних створено графік, що зображено на рисунку 2.5.

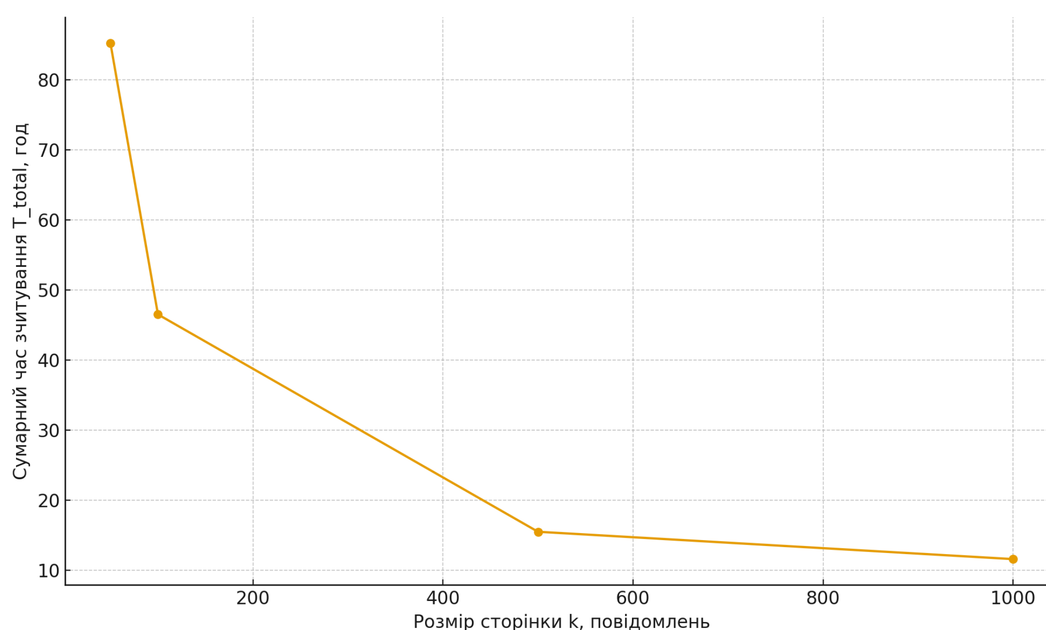

Рис. 2.5. Графік залежності часу повного зчитування

Цей графік демонструє, що в межах розглянутої моделі час повного доступу до історії зростає пропорційно до розміру таблиці, отже глибока історія діалогів, яка може досягати десятки мільйонів записів стає практично недоступною для аналітики через стандартний HTTP протокол без додаткових спеціалізованих інструментів.

2.3 Методи інтеграції моделей штучного інтелекту

2.3.1 Особливості роботи моделей ШІ як джерело інтеграційних обмежень

Сучасні мовні моделі штучного інтелекту, зокрема великі мовні моделі, працюють як імовірнісні перетворювачі тексту. На вхід вони отримують послідовність токенів, яка включає інструкцію для моделі та фрагмент даних, а на виході генерують нову послідовність токенів. Важливою властивістю таких моделей є відсутність вбудованої довготривалої пам’яті: кожен виклик виконується ізольовано, а рішення приймаються лише на основі поточного запиту, без доступу до попередніх звернень або стану бази даних. Це означає, що вся інформація, потрібна для побудови осмисленої відповіді, має бути включена до запиту у вигляді текстового контексту.

З технічного погляду інтеграція таких моделей у платформу чат-ботів передбачає організацію окремого контуру виклику: необхідно зібрати дані з кількох джерел, перетворити їх на текст, сформувати запит до моделі, дочекатися відповіді, обробити її та повернути у форматі, придатному для користувача або інших компонентів системи. Кожен із цих кроків супроводжується власними обмеженнями. Виникають затримки, пов’язані з мережею та часом інференсу моделі, існують обмеження на розмір контексту, встановлюються квоти та ліміти на кількість запитів, можуть траплятися помилки або відмови з боку постачальника моделі. Це створює ситуацію, коли логіка діалогу стає чутливою до зовнішніх факторів, які важко контролювати в межах самої платформи.

Додатковою складністю є те, що інтеграція моделей ШІ завжди відбувається на перетині різних технологічних доменів. З одного боку, є внутрішня структура платформи з її власними моделями даних, історією повідомлень і бізнес-контекстом.

З іншого – мовна модель, яка очікує на вхід лінійний текст і не має уявлення про структуру бази даних, сутності CRM чи специфіку конкретних сервісів. Будь-яка невідповідність між цими світами призводить до втрати інформації, двозначностей або помилкових інтерпретацій. У результаті стає очевидним, що для коректної інтеграції недостатньо просто «підключити» модель через API; необхідно вирішити низку концептуальних питань щодо того, які саме дані, у якій формі і в якому обсязі можуть бути нею коректно опрацьовані.

Окрему групу викликів формує вибір середовища виконання моделей. Якщо мовна модель розгортається локально, платформа отримує більше контролю над конфіденційністю даних і затримками, але водночас бере на себе відповідальність за інфраструктуру, масштабування та оновлення. Якщо ж модель використовується як зовнішній сервіс, навпаки, інфраструктурні питання перекладаються на провайдера, однак виникає залежність від його політик, тарифів і технічних обмежень. У будь-якому з цих варіантів інтеграція перетворюється на задачу балансування між якістю відповіді, витратами ресурсів, затримками та вимогами до безпеки даних.

У сукупності ці фактори показують, що складність інтеграції моделей ШІ визначається не тільки їх внутрішньою будовою, а й способом включення в архітектуру платформи. Для того, щоб мовна модель могла бути корисною в діалоговій системі, необхідно попередньо подолати розрив між її абстрактною текстовою природою та конкретними вимогами до зберігання, обробки й захисту даних у багатоплатформенному середовищі.

2.3.2 Проблема передачі контексту в багатоканальних діалогових системах

Однією з найскладніших проблем, що виникають під час інтеграції моделей штучного інтелекту в чат-бот платформи, є коректна передача контексту. Модель не має безпосереднього доступу ні до бази даних, ні до журналів подій, ні до історії взаємодій з користувачем, а працює лише з тим фрагментом інформації, який потрапив до запиту. Тому саме платформа несе відповідальність за те, щоб перетворити історію діалогу, дані про користувача, службові параметри та бізнес-контекст у компактний, але змістовний текстовий запит до моделі. Будь-яка

помилка на цьому етапі – пропуск важливої репліки, некоректна інтерпретація системного параметра чи надлишкове включення несуттєвих деталей – безпосередньо впливає на якість відповіді.

У рамках цієї роботи під контекстом розуміється сукупність даних, які необхідні моделі для побудови осмисленої відповіді. До нього належать попередні репліки діалогу, що зберігаються у сховищі повідомлень, інформація про користувача, службові параметри, що дозволяють ідентифікувати бота та сценарій, у якому він працює, а також бізнес-контекст, наприклад статус замовлення чи дані з CRM. На рівні платформи ці відомості розподілені між кількома сутностями й таблицями, однак для моделі ШІ вони мають бути зведені до єдиної лінійної послідовності тексту. Це перетворення завжди супроводжується втратою структури та примусовим спрощенням первинних даних.

Ситуація ускладнюється двома фундаментальними обмеженнями. Перше полягає в обмеженому розмірі контексту, який може опрацювати модель за один виклик. Кількість токенів є скінченною, тому повна історія взаємодії, що накопичується за тривалий час, не може передаватися цілком. Платформа змушена вибирати, які саме фрагменти діалогу та який обсяг супровідних даних слід включити до запиту, які репліки можна відсікати, узагальнювати або стискати, а які мають залишатися у первісному вигляді. Це перетворює задачу формування контексту на окрему оптимізаційну проблему, де будь-який вибір призводить до певних втрат інформації.

Друге обмеження пов’язане з багатоканальною природою сучасних платформ. Один і той самий користувач може взаємодіяти з ботом через різні канали, починаючи діалог у веб-чаті, продовжуючи його в мобільному месенджері та завершуючи в іншому інтерфейсі. Формально всі ці події можуть бути представлені уніфікованою моделлю повідомлень, проте з погляду мовної моделі вони все одно розпадаються на окремі фрагменти тексту, які необхідно правильно пов’язати між собою. Якщо платформа не враховує багатоканальність, модель або не бачить частину важливих реплік, або отримує надмірний, погано структурований контекст,

який вичерпує доступне вікно контексту й погіршує якість відповіді. Наглядно ця проблема зображена на рисунку 2.

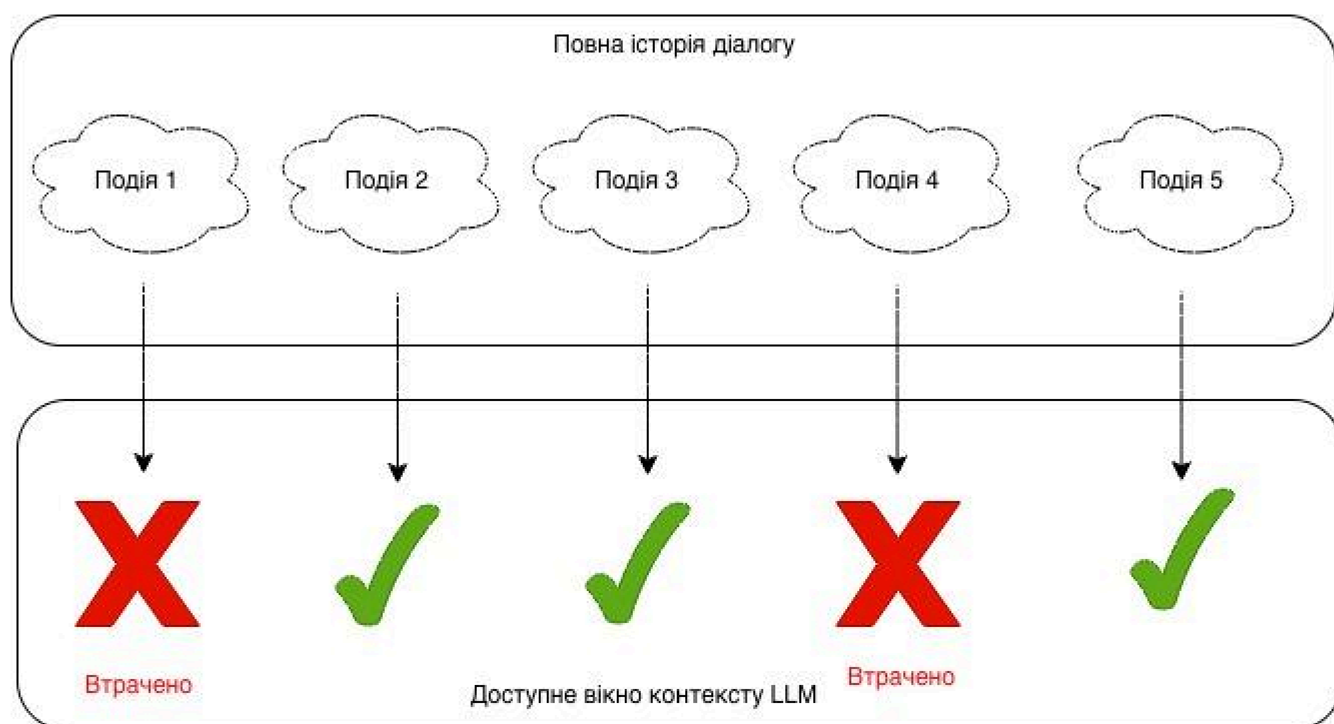

Рис. 2.5. Обмеження контекстного вікна LLM та втрати частини історії діалогу

У підсумку проблема передачі контексту полягає не тільки в механічному обмеженні обсягу даних, а й у необхідності процедури відбору, структурування та лінійного представлення інформації, яка від природи є багатовимірною та розподіленою. Саме від того, наскільки коректно платформа виконує цю роботу, залежить, чи зможе модель ШІ дати релевантну й послідовну відповідь в умовах реального багатоплатформенного середовища.

Висновок до розділу

У цьому розділі здійснено дослідження архітектурних підходів та методів проєктування систем для чат-ботів у контексті багатоплатформеного середовища. Проаналізовано вимоги до інтеграційного шару, показано обмеження традиційних монолітних рішень та виявлено, що проста мікросервісна декомпозиція без чітко

сформованої моделі даних і подій породжує власні проблеми. Особливу увагу приділено ролі самостійних чат-ботів як автономних модулів та впливу контейнеризації на ізоляцію середовищ і одночасне виникнення нових вузьких місць на рівні спільної інфраструктури. Сформульовані висновки дозволяють визначити практичні вимоги до інтеграційного шару, які знижують фрагментацію реалізацій і забезпечують більш кероване масштабування при підключенні нових каналів взаємодії.

На рівні логічної моделі було обґрунтовано доцільність уніфікованої моделі повідомлення Message, яка відокремлює світ зовнішніх API месенджерів від внутрішньої бізнес-логіки. Показано, що саме єдиний формат повідомлень дозволяє усунути фрагментацію сценаріїв, зменшити дублювання коду та створити передумови для централізованого керування діалогами. Разом із тим виявлено, що лінійна, повністю синхронна схема обробки подій(від прийому HTTP-запиту до формування відповіді) погано масштабується, ускладнює контроль послідовності подій та аналіз шляху конкретного повідомлення в системі, що вимагає переходу до більш гнучкої, подійно-орієнтованої архітектури.

Окремо досліджено проблему передачі контексту при інтеграції моделей штучного інтелекту. Показано, що обмеження розміру вікна контексту та фрагментація історії між різними каналами вимагають від платформи застосування продуманих стратегій відбору, стискання та конструювання інструкцій на основі уніфікованої моделі повідомлень. Розглянуто підхід до алгоримічного формування контексту та окреслено роль LLM як універсального компонента у поєднанні з вузькоспеціалізованими моделями, а також проаналізовано переваги й недоліки локальної інтеграції та інтеграції через зовнішні API. У підсумку це створює передумови для підвищення узгодженості історії діалогу та якості відповідей чат-бота в реальному багатоканальному середовищі.

РОЗДІЛ 3
ПРОЄКТУВАННЯ МОДЕЛЕЙ, МЕТОДІВ ТА АЛГОРИТМІВ ІНТЕГРАЦІЇ ЧАТ-БОТІВ У ПЛАТФОРМУ

3.1 Підходи до створення та підключення чат-ботів

3.1.1 Періодичне опитування API(Polling) як базовий спосіб підключення чат-ботів

Перед проєктуванням власного інтеграційного шару доцільно розглянути типові способи, за допомогою яких чат-боти підключаються до месенджерів у практичних реалізаціях. Найпростішим і найбільш інтуїтивним підходом є механізм періодичного опитування API платформи, відомий як polling. У цьому випадку саме бот виступає ініціатором взаємодії: він регулярно надсилає HTTP-запити до сервера месенджера з метою отримання нових подій, таких як повідомлення користувачів або службові оновлення.

На практиці цей підхід часто використовується на початкових етапах розробки. Процес створення Telegram-бота зазвичай починається з реєстрації через BotFather, отримання токена доступу та налаштування клієнтської бібліотеки для роботи з Telegram API. Після цього бот запускається у вигляді окремого процесу, який у нескінченному циклі виконує запити до API з певним інтервалом, отримує нові повідомлення та обробляє їх у межах власної логіки. Така модель є простою для розуміння й не потребує складної інфраструктури, оскільки не вимагає публічного серверного endpoint або додаткових мережевих налаштувань.

Однак уже на цьому рівні виявляються принципові обмеження polling-підходу. Час реакції бота безпосередньо залежить від інтервалу опитування: якщо він великий, користувач отримує відповідь із затримкою; якщо малий – різко зростає кількість запитів до API, що створює додаткове навантаження як на платформу месенджера, так і на серверну інфраструктуру бота. При переході від одного бота до десятків або сотень таких процесів це призводить до неконтрольованого зростання мережевого трафіку та ускладнює централізований моніторинг.

Весь цей процес зображено на рисунку 3.1, де кожен етап відображає етапи та дії, що потрібно виконати для успішного запуску такого бота на платформі Telegram.

Рис. 3.1. Етапи створення бота на платформі Telegram

У контексті розроблюваної платформи polling також підсилює проблему фрагментації, описану в розділі 2. Кожен бот має власні параметри опитування, власний цикл отримання подій, власну реалізацію обробки помилок і повторних запитів. Будь-яка зміна – наприклад, оновлення бібліотеки або коригування логіки отримання подій – потребує втручання в кожен окремий екземпляр. У результаті система швидко втрачає цілісність, а підтримка та масштабування перетворюються на трудомісткий процес.

3.1.2 Використання веб-хуків як подійної моделі підключення чат-ботів

На відміну від механізму періодичного опитування, у webhook-моделі ініціатором передачі подій виступає сама платформа обміну повідомленнями. У момент виникнення події (наприклад, надходження повідомлення від користувача або зміни статусу взаємодії) платформа формує подієвий запит і надсилає його на заздалегідь сконфігуровану адресу серверної частини бота. Такий підхід дозволяє

мінімізувати затримку між появою події та її обробкою, оскільки доставка здійснюється асинхронно та без необхідності періодичних запитів з боку бота.

Процес створення чат-бота у webhook-моделі починається зі створення облікового запису бота в екосистемі платформи та отримання ідентифікаційних параметрів доступу, які використовуються для автентифікації запитів. На наступному етапі розробник готує серверну частину бота, що містить публічну точку прийому подій. Ця точка повинна бути доступною ззовні, підтримувати захищене з'єднання та коректно обробляти вхідні HTTP-запити. Після цього URL точки прийому реєструється на стороні платформи як webhook-адреса, і з цього моменту всі події починають доставлятися безпосередньо до сервера бота.

Характерною особливістю webhook-підходу є те, що конфігурація доставки подій може включати додаткові параметри, зокрема перелік типів подій, які повинні надходити до системи. Це означає, що різні боти або екземпляри ботів можуть отримувати різний набір подій залежно від налаштувань, навіть якщо формально вони підключені за однаковою схемою. У багатоплатформенному середовищі така гнучкість з одного боку дозволяє оптимізувати потік даних, а з іншого — ускладнює уніфікацію інтеграції, оскільки внутрішній шар платформи повинен коректно обробляти різні комбінації подій.

З операційного погляду webhook-модель висуває підвищені вимоги до інфраструктури. Точка прийому подій повинна бути постійно доступною, оскільки її недоступність безпосередньо впливає на доставку повідомлень. Крім того, платформи зазвичай очікують швидкого підтвердження отримання події, що змушує розділяти процес прийому та подальшої обробки. Якщо сервер бота виконує складну логіку безпосередньо в HTTP-контексті, це може призводити до тайм-аутів і повторних доставок подій. У масштабованих системах ця особливість стає критичною, оскільки помилки одного компонента можуть впливати на стабільність усього контуру інтеграції.

У межах розроблюваної платформи webhook-підхід також сприяє накопиченню конфігураційної складності. При збільшенні кількості ботів зростає кількість endpoint-ів, параметрів доставки, ідентифікаційних ключів та правил

маршрутизації. Це ускладнює централізоване адміністрування й підсилює проблему фрагментації інтеграцій, описану в розділі 2. Таким чином, webhook-модель, хоча й забезпечує ефективну доставку подій, не усуває системних труднощів багатоботної архітектури, а лише переносить їх з рівня частоти запитів на рівень інфраструктурного управління.

3.1.3 Інтеграція чат-ботів через зовнішні сервісні платформи

Окремим класом рішень для побудови чат-ботів є використання спеціалізованих сервісних платформ керування діалогами, де значна частина логіки обробки повідомлень реалізується поза межами основної платформи. У такій моделі чат-бот не містить повної діалогової логіки, а виконує роль адаптера між платформою обміну повідомленнями та зовнішнім інтелектуальним сервісом. Однією з найбільш поширених платформ такого типу є Dialogflow, яка надає інструменти для розпізнавання намірів користувача, управління контекстом та формування відповідей на основі визначених сценаріїв.

Створення чат-бота з використанням Dialogflow зазвичай починається з ініціалізації проєкту в середовищі сервісної платформи. На цьому етапі формується агент, який представляє логічну модель бота та містить набір намірів, прикладів користувацьких фраз, параметрів і контекстів. Намір у такій системі описує певний тип взаємодії користувача з ботом і пов’язується з відповідними реакціями або діями. Таким чином, первинна “інтелектуальна” модель діалогу створюється не в коді, а у вигляді конфігурацій і сценаріїв, керованих через веб-інтерфейс або API сервісу.

Паралельно з налаштуванням агента створюється бот на стороні платформи обміну повідомленнями, який отримує необхідні ідентифікаційні параметри доступу. Серверна частина цього бота налаштовується таким чином, щоб приймати події від платформи, перетворювати їх у формат, придатний для сервісної платформи керування діалогами, та передавати у Dialogflow. Відповідь, сформована сервісом на основі визначеного наміру та активного контексту, повертається назад до серверної частини бота і надсилається користувачу через API платформи. Таким чином,

ланцюг обробки події включає кілька незалежних компонентів, кожен з яких відповідає за власну частину логіки. Приблизну архітектуру такого Dialogflow проєкта можна побачити на рисунку 3.2.

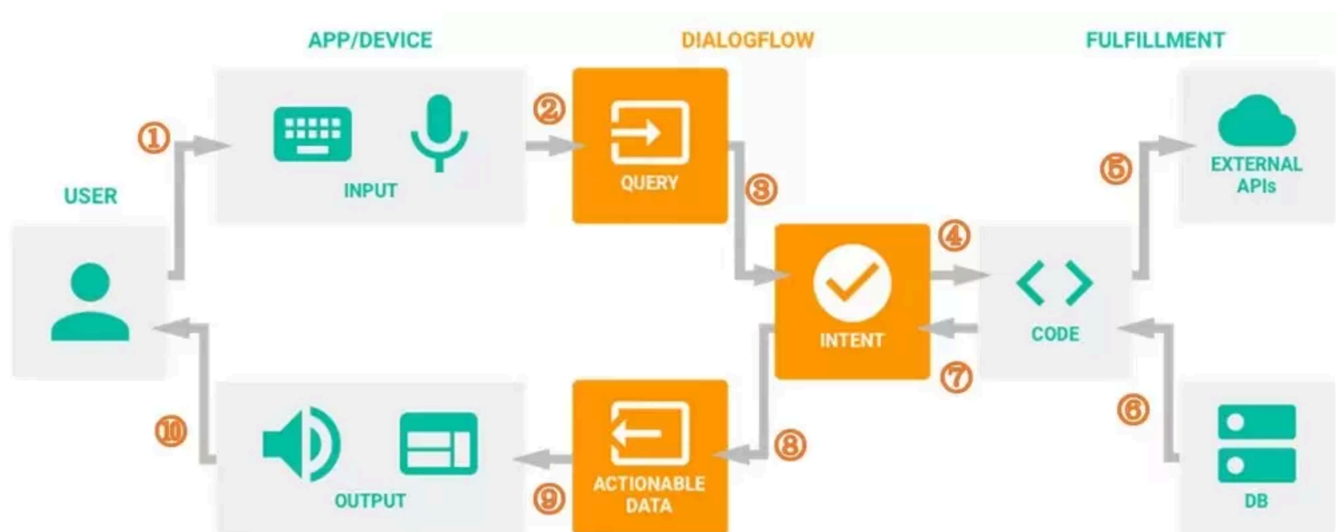

Рис 3.2. Архітектура Dialogflow додатку

Основною особливістю такої моделі є те, що стан діалогу та контекст взаємодії зберігаються і підтримуються зовнішнім сервісом. Dialogflow використовує власні механізми контекстів і сесій, які дозволяють пов’язувати послідовні репліки користувача та враховувати попередні дії. Це спрощує створення складних сценаріїв і дозволяє швидко модифікувати поведінку бота без внесення змін до серверного коду. Однак водночас виникає залежність від внутрішньої моделі даних сервісу, яка не завжди збігається з моделлю даних SaaS-платформи.

У багатоплатформенному середовищі така інтеграція створює додаткові труднощі. Вся інформація, якою оперує сервіс керування діалогами, повинна бути передана у вигляді запиту, тоді як значна частина бізнес-контексту, історії взаємодій і службових параметрів зберігається всередині платформи. Це вимагає постійної синхронізації між внутрішнім станом системи та зовнішнім сервісом і ускладнює підтримку єдиного джерела істини щодо стану діалогу. Крім того, будь-які зміни в структурі сценаріїв або контекстів сервісної платформи можуть вимагати адаптації інтеграційного шару, що знижує гнучкість системи в довгостроковій перспективі.

Таким чином, використання Dialogflow та подібних сервісних платформ дозволяє суттєво спростити початковий етап створення чат-ботів і швидко реалізувати складні діалогові сценарії. Водночас цей підхід переносить ключову частину логіки за межі платформи, посилює залежність від зовнішнього постачальника і ускладнює уніфікацію інтеграцій у багатоплатформенному середовищі. У контексті досліджуваної проблематики це підтверджує, що сервісна інтеграція не усуває фрагментацію, а лише змінює її форму, що зумовлює необхідність пошуку централізованих підходів до керування ботами та інтеграційними процесами.

3.1.4 Уніфікований інтеграційний шар та автономні боти в керованому контейнерному середовищі

Результати аналізу підходів Polling, Webhook та сервісної інтеграції показали, що кожен із них дозволяє організувати доставку подій до бота, однак жоден не усуває системну проблему багатоплатформенного середовища. У міру зростання кількості каналів та сценаріїв інтеграції виникає фрагментація на рівні протоколів, форматів подій, налаштувань, правил маршрутизації та експлуатаційних процесів. У підсумку платформа отримує множину ізольованих реалізацій ботів, кожна з яких розвивається за власними правилами, а будь-яка зміна або збій у такій системі набуває ефекту масштабу. Це створює потребу у цілісному, платформенному вирішенні, яке одночасно зменшує різнорідність каналів, підвищує керованість та забезпечує єдині правила обробки подій.

У межах даної роботи пропонується вирішувати зазначену проблему шляхом побудови уніфікованого інтеграційного шару, що виступає центральним шлюзом між зовнішніми платформами та внутрішньою логікою SaaS-системи. Вхідні повідомлення з різних каналів приводяться до єдиної моделі, а подальша маршрутизація й керування станом виконуються поза межами транспортного рівня конкретного месенджера. Архітектурно така ідея узагальнена на рисунку 3.3, де інтеграційний шар реалізує абстракцію бізнес-логіки від специфіки протоколів і форматів подій окремих платформ.

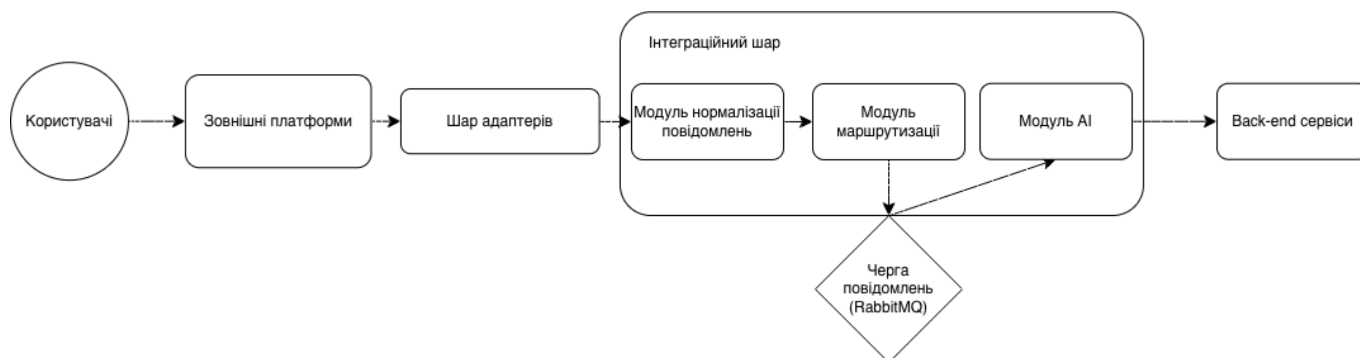

Рис. 3.3. Концептуальна архітектура багатоплатформенного інтеграційного шару.

З практичного погляду це означає, що додавання нового каналу не повинно змінювати внутрішні механізми платформи, а лише доповнювати систему відповідним адаптером входу, який перетворює події у спільний формат. При цьому важливо зберегти автономність окремих ботів як виконавчих модулів, оскільки саме вона дозволяє розділити відповідальності та зменшити взаємозалежність компонентів. У такій організації кожен бот може обслуговувати певний канал або окремий бізнес-сценарій, залишаючись незалежним у своєму життєвому циклі, але при цьому працюючи не з "сирими" подіями платформи, а з уніфікованими повідомленнями інтеграційного шару. Взаємодія автономних ботів із централізованим шаром узагальнена на рисунку 3.4.

Додатковою проблемою є фрагментація аналітики та моніторингу. Події, пов'язані з діалогами, розподіляються між кількома ботами, і для повноцінного аналізу взаємодії з користувачами потрібно агрегувати дані з різних джерел. Це ускладнює оцінку загальної якості сервісу, виявлення аномалій та порівняння ефективності ботів між собою. За відсутності єдиного сховища метрик і централізованого журналювання зростає ризик неповноти даних та виникнення різночитань у показниках, отриманих з різних платформ. Тому на практиці доцільним є впровадження централізованого моніторингу та кореляції подій, що дозволяє відстежувати повний шлях звернення користувача і формувати узгоджену аналітичну картину.

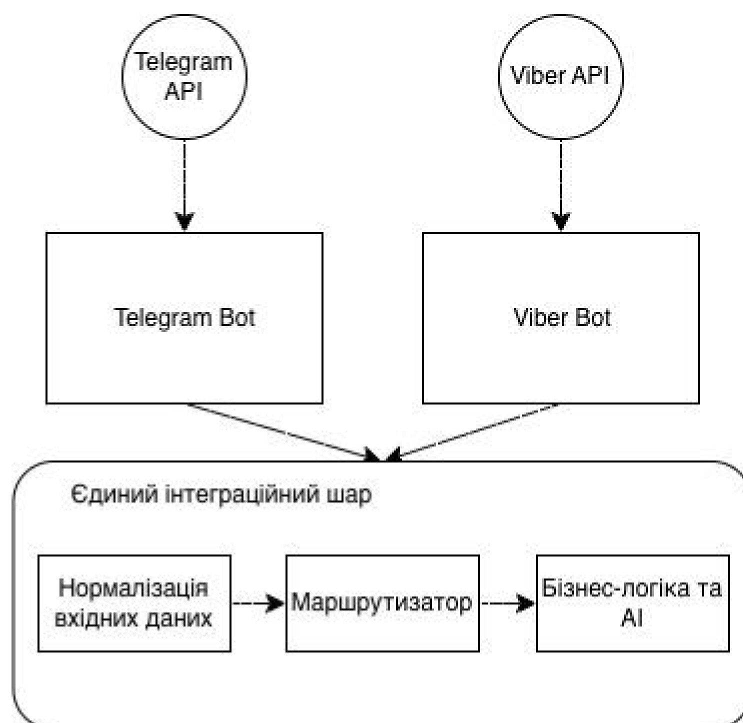

Рис. 3.4. Схема взаємодії самостійних ботів з інтеграційним шаром

Технічною основою для забезпечення автономності та ізоляції виконання таких ботів у рамках платформи обирається контейнеризація. Контейнер виступає як одиниця, що фіксує середовище виконання конкретного бота, включно з необхідними бібліотеками, SDK, конфігураціями та мережевими налаштуваннями, і дозволяє запускати, зупиняти або масштабувати його незалежно від інших компонентів. Такий підхід підтримує ключові вимоги до системи: усуває конфлікти залежностей між різними ботами, спрощує оновлення окремих компонентів і дозволяє масштабувати навантаження точково, додаючи екземпляри лише тих ботів, які обслуговують перевантажений канал. Узагальнена схема розміщення контейнеризованих ботів у спільній інфраструктурі інтеграційного шару подана на рисунку 3.5.

У межах розроблюваної платформи керування життєвим циклом контейнеризованих ботів передбачається здійснювати програмно за допомогою Python SDK, що надає інтерфейси для взаємодії з контейнерним середовищем. Це дозволяє інтегрувати операції запуску, зупинки, оновлення та масштабування ботів безпосередньо в логіку платформи, не покладаючись на ручне адміністрування.

Такий підхід створює основу для автоматизації управління інтеграціями та узгодженого контролю стану ботів у багатоплатформенному середовищі.

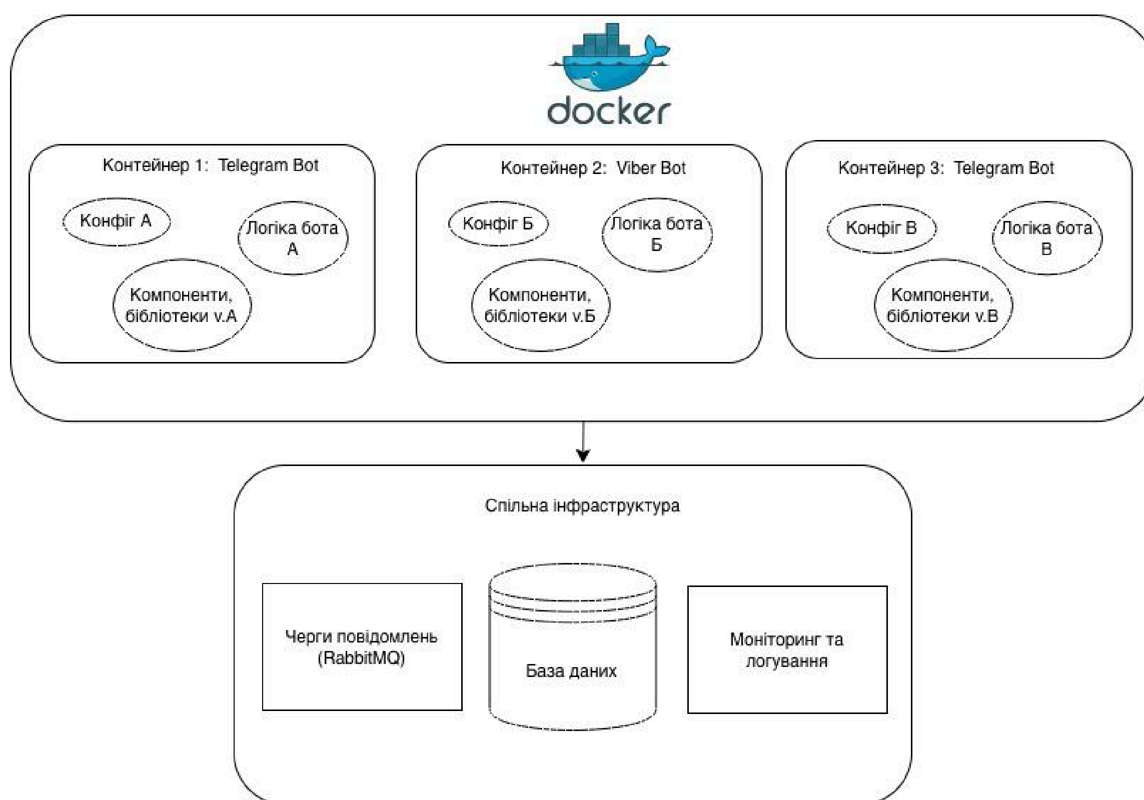

Рис. 3.5. Схема розгортання та взаємодії контейнеризованих чатботів

Разом із тим контейнеризація не знімає всіх ризиків, а переносить частину складності на рівень спільної інфраструктури. Незалежність ботів виявляється умовною, оскільки вони користуються спільними ресурсами — сховищами даних, системами обміну повідомленнями, конфігураційними механізмами, засобами моніторингу та логування. Перевантаження або збій будь-якого з цих компонентів впливає на всю систему, концентруючи потенційні точки відмови саме у спільному інфраструктурному контурі. Крім того, зі зростанням кількості контейнерів зростає операційна складність керування їхнім життєвим циклом: необхідно підтримувати контроль версій, узгоджувати конфігурації, відстежувати використання ресурсів і забезпечувати спостережуваність. У разі відсутності автоматизованих механізмів оркестрації та централізованого моніторингу такі системи стають важкими в підтримці й схильними до деградації при масштабуванні.

Таким чином, запропоноване вирішення проблеми фрагментації базується на поєднанні трьох взаємодоповнювальних принципів: централізації інтеграційних функцій у вигляді уніфікованого шару, автономності ботів як виконавчих модулів та ізоляції їхнього середовища виконання шляхом контейнеризації. Разом ці принципи дозволяють переходити від “набору несумісних інтеграцій” до керованої платформи, де підключення нових каналів і розвиток сценаріїв відбуваються без руйнівного зростання залежностей. Водночас на рівні проєктування необхідно враховувати, що контейнери не усувають потреби в централізованому контролі спільної інфраструктури, тому подальші підрозділи розділу 3 мають бути спрямовані на формалізацію механізмів керування життєвим циклом ботів та забезпечення стабільності спільних сервісів платформи.

3.2 Проєктування структури бази даних в системі чатботів

3.2.1 Формалізація обсягів та оцінка зберігання повідомлень у базі даних

Для подальшого аналізу архітектури бази даних доцільно формалізувати, які саме дані про повідомлення мають зберігатися в системі. Виходячи з вимог, сформульованих в попередньому підрозділі, повідомлення розглядається як кортеж, який можна представити в форматі кортежа m, що зберігає такі поля як id, bot_id, contact_id, timestamp, msg_id, is_from_bot, content, attachments, raw, extra. При проєктування схеми бази даних для СУБД на всі ці поля ми визначаємо фіксовану довжину та поля змінного розміру. До першої групи належать ідентифікатори об’єктів, часові мітки та булеві значення. Орієнтовна структура за типами може бути наведена в таблиці 2.1.

Таблиця 3.1.

Орієнтовна структура запису повідомлення в базі даних

Поле	Тип даних	Орієнтований розмір
id	UUID	16 байт
bot_id	INT	4 байти

Продовження таблиці 3.1.

contact_id	INT	4 байти
timestamp	TIMESTAMP	8 байт
message_id	TEXT / VARCHAR	змінна довжина
is_from_bot	BOOL	1 байт
content	TEXT	змінна довжина
attachments	JSONB	змінна довжина
raw	JSONB	змінна довжина
extra	JSONB	змінна довжина

Фізичний розмір одного такого кортежу повідомлення можна подати в вигляді наступної формули:

$$S_{msg} = S_{hdr} + S_{fixed} + S_{text/json}, \quad (3.1)$$

де S_{hdr} – службовий заголовок рядка(службові байти, інформація про довжину та стан запису, вирівнювання);

S_{fixed} – сумарний розмір полів фіксованої довжини;

$S_{text/json}$ – сукупний розмір полів змінної довжини разом із їхнім внутрішнім заголовком.

Розмір частини з фіксованою довжиною можна представити в вигляді наступної формули:

$$S_{fixed} = S_{uuid} + 2S_{int} + S_{ts} + S_{bool}, \quad (3.2)$$

де S_{ts} відповідає значенню timestamp з таблиці 3.1.

Підставивши числові значення, ми отримуємо наступний результат:

$$S_{fixed} = 16 + 2 \cdot 4 + 8 + 1 = 33 \text{ байти.}$$

Що ж до полів з змінної довжини доцільно ввести середні оцінки, ґрунтуючись на очікуваних сценаріях використання:

1. середня довжина ідентифікатора повідомлення провайдера L_{msg_id} буде становити 32 байти;
2. середня довжина тексту повідомлення $L_{content}$ буде становити 512 байт з урахуванням коротких і довгих реплік;
3. середня обсяг опису вкладень $L_{attachments}$ буде становити 256 байтів;
4. середній обсяг сирого повідомлення L_{raw} буде дорівнювати 1024 байти, оскільки сам payload містить повну структуру події від платформи;
5. середній обсяг додаткових даних L_{extra} буде становити 256 байт.

Кожне текстове або JSON-поле має власний внутрішній заголовок(кілька службових байтів), тому розмір змінної частини можна наближено подати в вигляді наступної формули:

$$S_{text/json} = (L_{msg_id} + \delta) + (L_{content} + \delta) + (L_{attachments} + \delta) + (L_{raw} + \delta) + (L_{extra} + \delta) \quad (3.3)$$

де δ – середній накладний обсяг додаткових байтів для одного поля змінної довжини. В подальших розрахунках можна прийняти $\delta \approx 4$ байти. Підставивши всі вищезазначені значення для кожного поля і отримаємо наступний результат:

$$S_{text/json} \approx (32 + 4) + (512 + 4) + (256 + 4) + (1024 + 4) + (256 + 4) \approx 36 + 516 + 260 + 1028 + 260 \approx 2100 \text{ байт}.$$

Якщо прийняти службовий заголовок рядка S_{hdr} на рівні приблизно 24 байтів, повний розмір кортежу приблизно становить:

$$S_{msg} \approx 24 + 33 + 2100 \approx 2157 \text{ байт} \approx 2,1 \text{ КБ}.$$

Отримана результат не претендує на точність до байта, однак дає важливий для подальшого планування висновок: середній запис про повідомлення з повним сирим payload та додатковими метаданими займає трохи більше аніж 2 КБ.

Маючи оцінку розміру одного кортежу повідомлення, можна перейти до наступного кроку дослідження – оцінки максимальної можливої кількості записів у таблиці повідомлень за обмеженого обсягу дискового сховища.

Нехай під базу даних системи відведено логічний том місткістю C_{disk} байт. Частина цього простору буде зайнята службовими структурами СУБД, тому введемо коефіцієнт корисної місткості α, який показує, яка частка диска реально може бути використана під таблицю messages.

$$C_{eff} = \alpha \times C_{disk}, \ 0 < \alpha < 1 \qquad (3.4)$$

Максимальна можлива кількість кортежів у таблиці повідомлень тоді визначається за наступним співвідношенням:

$$N_{max} = \left\lfloor \frac{C_{eff}}{S_{msg}} \right\rfloor = \left\lfloor \frac{\alpha \times C_{disk}}{S_{msg}} \right\rfloor \qquad (3.5)$$

Для конкретизації припустимо, що під зберігання історії діалогів виділяється логічний том обсягом:

$$C_{disk} = 200 \text{ ГБ} = 20 \times 2^{30} \approx 2,147 \times 10^{11} \text{байт},$$

а під саму таблицю планується відводити приблизно $\alpha = 0,7$ цього обсягу. Тоді ефективна місткість можна розраховувати за формулою 3.4 з наступним результатом:

$$C_{eff} = 0,7 \times 2,147 \times 10^{11} \approx 1,503 \times 10^{11} \text{ байт.}$$

Підставляючи сюди оцінку розміру одного кортежу, отримаємо

$$N_{max} \approx \left[\frac{1,503 \times 10^{11}}{2,157}\right] \approx 6,97 \times 10^{7},$$

тобто, орієнтовно близько 70 мільйонів записів в таблиці повідомлень. Ця величина є теоретичною верхньою межею за прийнятих припущень щодо структури кортежу та частки диска, доступної таблиці. На практиці фактичний ліміт буде дещо нижчим через додаткові індекси, фрагментацію сторінок, резервні області та можливі зміни середнього розміру полів content, attachments, raw і extra. Водночас отриманий порядок величини дає змогу оцінити, що за повного збереження payload та метаданих таблиця повідомлень досить швидко досягає десятків мільйонів рядків, що безпосередньо впливає на подальші рішення щодо архівування.

3.2.2 Уніфікована модель повідомлення в багатоплатформеній системі

Першим кроком до впорядкованої обробки подій у багатоплатформеній системі чат-ботів є формалізація поняття повідомлення як базової сутності платформи. На рівні окремих месенджерів це поняття вже визначене: кожен канал має власну схему даних, власні ідентифікатори та власну логіку подання діалогу. Проте для SaaS-платформи, яка одночасно інтегрується з кількома каналами та обслуговує різні сценарії взаємодії, такого локального визначення є недостатньо.

Саме тому в межах даної роботи вводиться поняття уніфікованої моделі повідомлення як внутрішньої сутності платформи. Вона не копіює жоден із зовнішніх форматів «як є», а задає власний, незалежний від конкретного провайдера

спосіб опису будь-якої події типу повідомлення. Усі ключові компоненти системи – керування діалогами, зберігання історії, аналітика, інтеграція зі сторонніми сервісами – працюють не з сирими payload'ами месенджерів, а саме з цією уніфікованою моделлю. Перетворення зовнішніх форматів у внутрішній виконується на етапі інтеграції, після чого події циркулюють системою в єдиному представленні.

Загальна структура основних моделей платформи, зокрема сутностей бота, контакту та повідомлення, подана на рисунку 3.6.

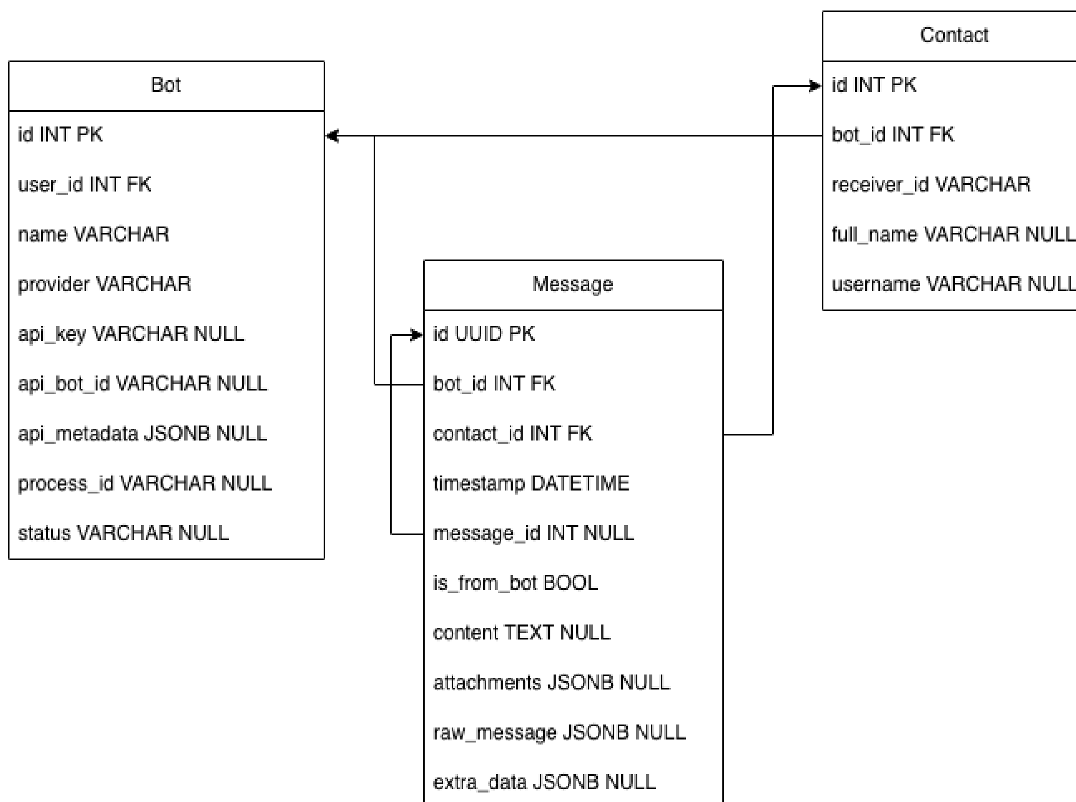

Рис. 3.6. Схема основних моделей в системі

У запропонованій архітектурі модель повідомлення реалізується у вигляді сутності Message, для якої в базі даних створюється окрема таблиця messages. Вона містить внутрішній унікальний ідентифікатор на основі UUID, посилання на відповідного бота та контакт, часову мітку, текстовий вміст, структуру вкладень, а також поля для збереження сирого повідомлення та додаткових метаданих конкретного провайдера. Таким чином, одна сутність поєднує бізнесовий зміст діалогу з повним технічним слідом події.

На перший погляд таке рішення може здатися надмірно складним, однак саме уніфіковане представлення дозволяє усунути ключові проблеми багатоплатформенного середовища. У типовій ситуації платформа одночасно отримує події з кількох джерел: вхідні та вихідні повідомлення, редагування, службові нотифікації, зміни статусів чатів. Для кожного такого типу подій зовнішні платформи пропонують власні, часто громіздкі та специфічні структури даних. Без уніфікації платформа змушена «мислити» в термінах цих форматів, що призводить до жорсткої прив’язки діалогової логіки до конкретних API.

Уніфікована модель повідомлення пропонує інший підхід. Замість адаптації сценаріїв під кожен формат платформа вводить власну внутрішню «мову» опису подій. Усі зовнішні повідомлення на рівні інтеграції приводяться до формату Message, а на рівні бізнес-логіки використовується виключно ця модель. Таким чином, світ інтеграції з месенджерами та світ діалогової логіки чітко розділяються: перший може змінюватися та розширюватися, тоді як другий залишається стабільним і незалежним від конкретних каналів.

3.2.3 Проблематика обробки подій у багатоплатформеній діалоговій системі

Після введення уніфікованої моделі повідомлення логічним наступним кроком є аналіз того, як ці повідомлення фактично “протікають” крізь систему. Якщо розглядати модель Message як статичний опис, усе виглядає доволі просто: є таблиця messages, у ній зберігаються записи з ідентифікаторами бота й контакту, часовими мітками, вмістом і метаданими. Однак у реальній роботі платформи ця таблиця заповнюється не вручну, а внаслідок безперервного потоку подій з боку користувачів і самих ботів.

У найпростішому уявленні можна сказати, що кожна подія “Нове повідомлення” перетворюється на новий рядок у таблиці messages і паралельно запускає певну логіку обробки: потрібно оновити стан діалогу, згенерувати відповідь, можливо, зафіксувати дані для аналітики або CRM. Саме тут стає помітно, що однієї лише грамотно спроєктованої моделі повідомлення недостатньо. Важливо

розуміти, як і в якому порядку ця модель використовується в умовах багатоплатформеного, розподіленого середовища.

Цей лінійний сценарій добре працює, доки система невелика: кілька ботів, обмежена кількість контактів, невисока інтенсивність повідомлень. Однак у міру росту платформи з'являються нові боти, нові канали, збільшується кількість користувачів, сценарії діалогів стають складнішими та потребують звернень до зовнішніх сервісів. Навіть якщо уніфікована модель повідомлення вже впроваджена, стає очевидно, що такої простоти недостатньо.

Першим джерелом проблем є жорстка синхронність обробки. Один HTTP-запит фактично тягне за собою весь ланцюжок дій: приймання, валідацію, нормалізацію, запис до бази, завантаження історії діалогу, оновлення стану, генерацію відповіді, формування вихідного payload і надсилання назад до платформи. Кожен із цих кроків має власну вартість за часом і власні можливі точки відмови. Якщо, наприклад, діалогова логіка у певний момент вимагає звернення до зовнішньої CRM-системи або довгої генерації відповіді AI-моделлю, один такий важкий сценарій для конкретного користувача здатен на певний час заблокувати обробку всіх подій, що потрапили в той самий потік виконання. У практиці це проявляється як накопичення запитів, зростання часу відгуку та ризик повторних доставок з боку платформ, які не отримали підтвердження вчасно. Таку лінійність представлено на рисунку 3.7.

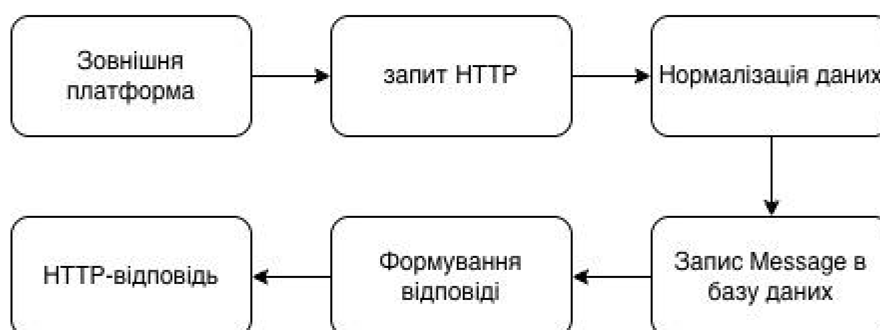

Рис. 3.7. Лінійна схема обробки події "Нове повідомлення"

Також виникають проблеми з послідовністю подій. У реальному середовищі

два повідомлення від одного користувача можуть прийти на сервер у «переплутаному» порядку через мережеві затримки чи балансувальник. У таблиці messages вони будуть впорядковані за timestamp, але обробка діалоговим менеджером могла відбутися в іншій послідовності: друга подія оброблена раніше, ніж перша. Без додаткових механізмів це призводить до того, що стан діалогу оновлюється в нелогічному порядку, і хоча записи в базі виглядають правильно, реальний хід подій для користувача був іншим.

При спробі масштабувати систему горизонтально, запустивши кілька інстансів застосунку, різні екземпляри можуть паралельно приймати події для одного контакту, і узгодженість стану доводиться забезпечувати за рахунок блокувань в базі даних або складних транзакцій. Щоб структуровано зафіксувати слабкі місця, доцільно уявити цей базовий процес як умовний алгоритм лінійної обробки, що зображено на рисунку 3.8.

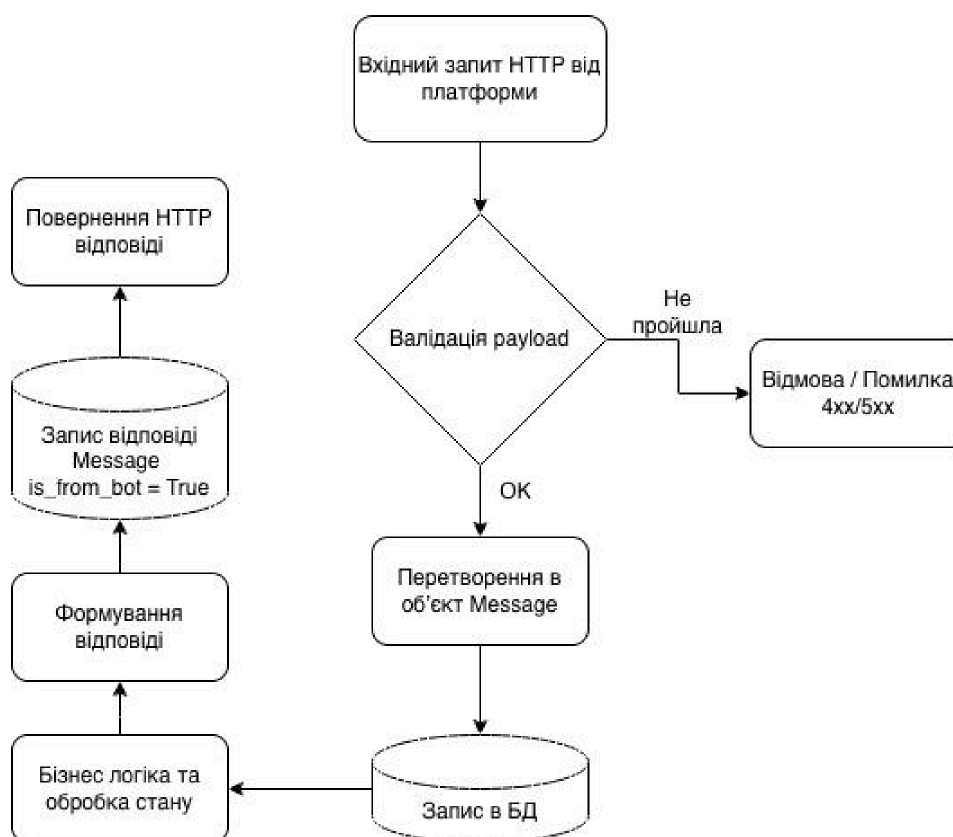

Рис. 3.8. Алгоритм лінійної обробки події «нове повідомлення» у межах одного процесу

У такому вигляді стає видно, що навіть за наявності уніфікованої моделі повідомлення система залишається сильно зв'язаною: прийом подій, їх запис, обробка стану й формування відповіді виконуються як єдиний, синхронний ланцюг. Це ускладнює масштабування, збереження послідовності діалогу та прозорий моніторинг. Саме ці обмеження і формують постановку задачі для подальших підрозділів: знайти спосіб розв'язати модель Message із конкретного життєвого циклу, запровадити проміжний рівень для керування потоком подій та забезпечити більш гнучку, подійну архітектуру обробки.

3.3 Моделі організації черг та обробки подій на основі RabbitMQ

3.3.1 Модель черг та маршрутизації подій у RabbitMQ

Попередній підрозділ показав, що навіть за наявності уніфікованої моделі Message система залишається вразливою через лінійну, синхронну схему обробки подій. У такій схемі приймання події, її валідація, нормалізація, запис у базу даних, формування відповіді та повернення її у відповідний канал фактично виконуються як єдиний ланцюг у межах одного потоку виконання. Це призводить до накопичення запитів під навантаженням, ускладнює масштабування, підвищує ризик повторної доставки подій з боку платформ і робить процес діагностики залежним від ручного зіставлення логів. Для усунення цих обмежень у платформі вводиться проміжний рівень керування потоком подій у вигляді брокера повідомлень, який переводить систему з синхронної моделі в подієву та асинхронну [35].

У межах даної роботи як базовий механізм подієвого обміну обрано RabbitMQ, який виступає шиною подій між ботами, інтеграційним шаром та сервісним ядром. Використання черг дозволяє відокремити приймання повідомлення від його подальшої обробки, а також забезпечити кероване накопичення подій у моменти пікових навантажень. У такому підході ботові модулі не взаємодіють безпосередньо з базою даних або з сервісами бізнес-логіки: вони публікують події у брокер, а сервісне ядро асинхронно споживає їх, виконує нормалізацію в модель Message, зберігає результат та ініціює необхідні дії (формування відповіді, звернення до

зовнішніх сервісів, оновлення контексту). Після завершення обробки відповідь повертається у брокер і доставляється назад до відповідного бот-модуля, який вже виконує відправлення у зовнішній канал. Така організація знімає жорстку прив’язку до часових обмежень HTTP-викликів та перетворює обробку повідомлень на керований конвеєр, де кожен етап може масштабуватися та відновлюватися незалежно.

Важливим елементом цієї моделі є логічне розділення потоків подій. Для цього RabbitMQ застосовується не лише як “одна черга на все”, а як система маршрутизації, де різні типи повідомлень проходять різними каналами. Вхідні події від зовнішніх платформ надходять у виділений вхідний контур, тоді як сформовані відповіді передаються через окремий вихідний контур. Поділ потоків дозволяє уникнути змішування різних типів навантаження, а також забезпечує більш прозорий моніторинг: видно, де саме накопичуються повідомлення і на якому етапі виникають затримки. Сам принцип побудови такого конвеєра доцільно подати графічно і ознайомитись з ним можна на рисунку 3.9.

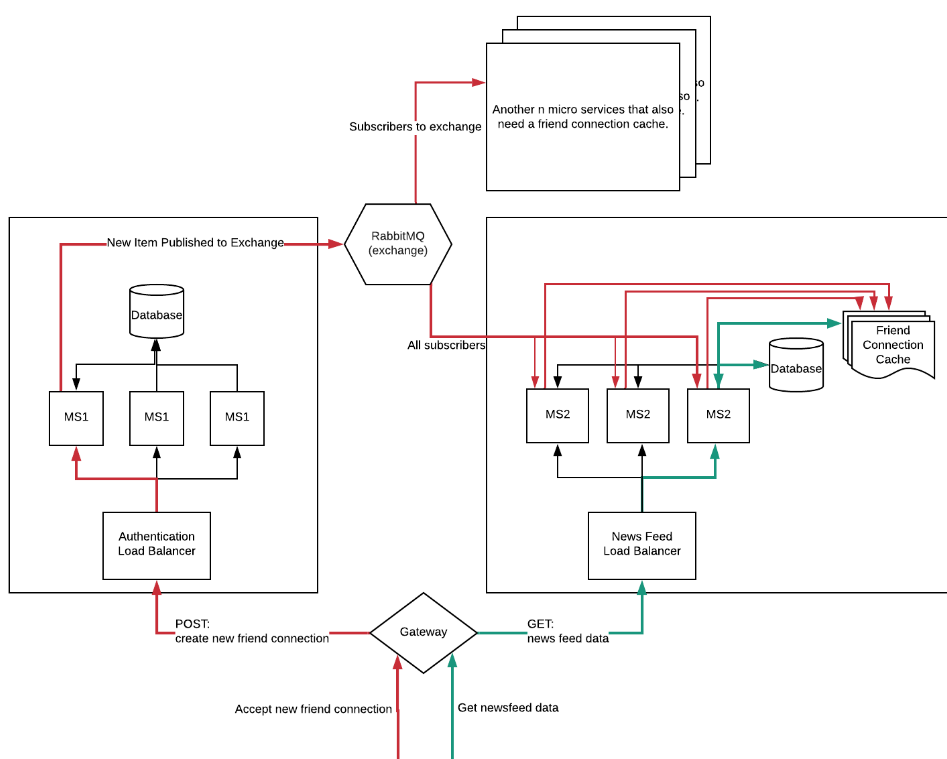

Рис. 3.9. Подієвий конвеєр обробки повідомлень на основі RabbitMQ

На рисунку показано, що модулі інтеграції з каналами публікують події у вхідний потік черг, після чого сервісне ядро споживає їх, формує уніфіковану модель Message та виконує обробку. Результати обробки передаються у вихідний потік, звідки окремі модулі доставки відправляють відповіді користувачам у відповідні платформи.

3.3.2 Порівняльне обґрунтування вибору RabbitMQ замість Redis як брокера повідомлень

У якості брокера повідомлень для платформи можна було розглядати як Redis, так і RabbitMQ. Проте для підтримки складнішої маршрутизації подій, гарантії доставки та чіткого розділення потоків було обрано саме RabbitMQ. Його модель, що базується на exchange-об'єктах, чергах та routing key, природно підтримує сценарії типу "одне повідомлення – кілька споживачів", групування черг за функціональними ролями й організацію окремих каналів для вхідних, вихідних та службових подій.

Крім того, RabbitMQ надає розширені механізми підтвердження доставки(ack/nack), персистентності та настроювання політик черг, що є дуже важливим для платформи, де втрата повідомлення означає втрату частини історії діалогу. Redis добре підходить для кешування, швидкого доступу до невеликих структур даних чи простих механізмів pub/sub, однак для складних сценаріїв маршрутизації з персистентними чергами й контролем споживачів Redis-підхід вимагав би додаткового прикладного коду, який RabbitMQ вже надає з коробки.

3.3.3 Алгоритм двонапрямленого каналу команд та статусів

Окрім основного потоку діалогових повідомлень у багатоботній платформі необхідний окремий контур керування, який дозволяє централізовано впливати на життєвий цикл виконавчих модулів та одночасно отримувати зворотний зв'язок про їхній стан. У практичних умовах система не може розглядати бот як "статичний процес", який завжди працює коректно: боти можуть аварійно завершуватися, потребувати перезапуску, оновлення конфігурації, зміни режиму роботи або тимчасового відключення. Якщо такі дії виконуються вручну або через зовнішні

адміністративні канали, платформа втрачає керованість і спостережуваність, що прямо суперечить вимогам масштабованості та відмовостійкості. Саме тому доцільно вводити окремий двонапрямлений канал команд та статусів, організований тим самим подієвим способом, що й обробка повідомлень, але логічно відокремлений від діалогового трафіку.

Двонапрямлений канал будується як регулярний алгоритм обміну службовими повідомленнями між керуючим контуром платформи та бот-модулями. У загальному випадку керуюча сторона формує команду, адресовану конкретному боту або групі ботів, і публікує її у виділений канал. Бот-модуль, який підписаний на відповідні команди, приймає повідомлення, виконує його у межах власного середовища виконання та формує статусну відповідь. Повернення статусу є принциповим: команда без підтвердження виконання не дає платформи розуміння, чи була дія здійснена, чи завершилась помилкою, чи взагалі була отримана. Тому алгоритм передбачає, що кожна команда породжує один або кілька статусних сигналів, які містять результат виконання та базові метадані.

З погляду логіки керування цей канал вирішує відразу кілька задач. Він дозволяє реалізувати централізоване управління ботами без прямого доступу до їх процесів, уніфікує спосіб передачі керуючих дій незалежно від конкретної платформи месенджера, а також формує основу для моніторингу станів, оскільки статусні повідомлення можуть агрегуватися в системі спостережуваності. Важливо, що цей канал має бути незалежним від основного потоку повідомлень користувача, щоб пік діалогового навантаження не блокував керуючі команди, а сервісні події керування не "засмічували" черги обробки повідомлень.

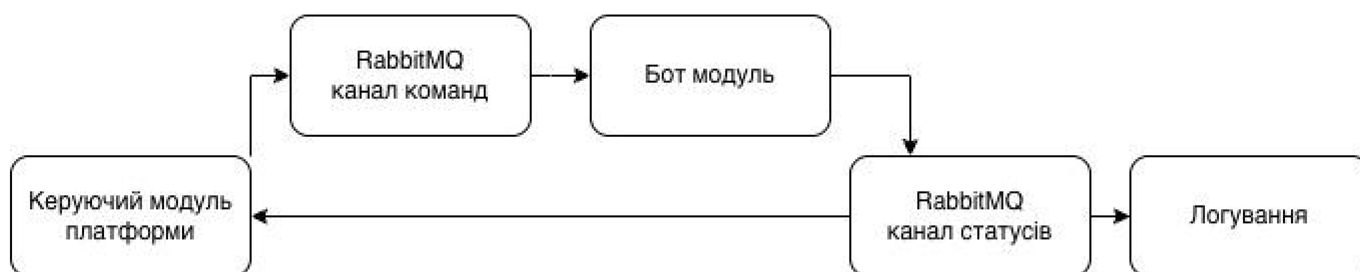

Рис. 3.10. Схематичне представлення двонапрямленого каналу в RabbitMQ

Алгоритмічно обмін у двонапрямленому каналі доцільно розглядати як послідовність кроків. Керуючий модуль платформи ініціює подію керування, публікуючи команду із зазначенням адресата, типу дії та параметрів виконання. Бот-модуль приймає команду, переходить у режим виконання, після чого формує статусний результат. Цей результат може бути проміжним(наприклад, “прийнято”, “виконується”) або фінальним(“виконано”, “помилка”), що особливо важливо для операцій, які не завершуються миттєво. Платформа, отримавши статус, фіксує його у журналі керування та, за потреби, ініціює наступну дію: повторну спробу, сповіщення адміністратора або автоматизоване масштабування.

3.4 Архітектурні рішення інтеграції LLM

3.4.1. Роль великих мовних моделей у багатоплатформеній системі чат-ботів

Інтеграція великих мовних моделей у багатоплатформену чат-бот систему не може розглядатися як ізольоване підключення зовнішнього сервісу генерації тексту. У межах SaaS-платформи LLM виступає як складова частина загального конвеєра обробки подій, тісно пов’язана з моделлю Message, механізмами збереження історії діалогу та асинхронною архітектурою обробки повідомлень. Тому проєктування інтеграції ШІ має спиратися не лише на можливості самої моделі, а й на особливості платформи, яка формує запити, керує контекстом і контролює життєвий цикл кожної інтелектуальної операції.

У розроблюваній архітектурі великі мовні моделі не розглядаються як автономні агенти, що зберігають власний стан діалогу або приймають рішення незалежно від платформи. Навпаки, LLM виступає як обчислювальний компонент, який реагує на запит, сформований платформою, і повертає результат у вигляді текстової відповіді або структурованих даних. Усі питання, пов’язані зі збереженням історії, контролем послідовності діалогу, багатоканальністю та бізнес-контекстом, залишаються на стороні платформи.

Такий підхід є принциповим для SaaS-середовища, де одна й та сама модель може обслуговувати десятки або сотні різних ботів, кожен з яких має власну логіку,

сценарії та правила взаємодії з користувачами. LLM у цьому випадку виконує роль універсального інтерпретатора та генератора, тоді як платформа визначає коли, для кого і з яким контекстом вона використовується. Це дозволяє уникнути жорсткої прив'язки діалогової логіки до конкретної моделі та зберегти контроль над поведінкою системи в цілому.

Крім генерації відповідей, LLM може залучатися до виконання допоміжних інтелектуальних задач: перефразування, узагальнення історії діалогу, витягування сутностей або пояснення рішень. Проте в усіх випадках модель працює виключно з тим набором даних, який їй передає платформа, що підкреслює ключову роль архітектури інтеграції.

3.4.2 Формування контексту для LLM на основі уніфікованої моделі

Однією із найважчих проблем інтеграції моделей штучного інтелекту в чат-бот платформах є коректна передача контексту. Модель не бачить всю базу даних і не має вбудованої пам'яті про попередні взаємодії з користувачем – вона оперує лише тим фрагментом інформації, який отримує в запиті(інакше prompt). Тому саме платформа відповідає за те, щоб перетворити історію діалогу, метадані об'єктів Message, профіль контакту та інші релевантні дані у компактний, але змістовний текстовий запит до моделі.

Під контекстом у рамках цієї роботи розуміється сукупність даних, які необхідні моделі для побудови осмисленої відповіді: попередні репліки діалогу, що збережені у таблиці messages, інформація про користувача, службові параметри, щоб розуміти, який саме бот відповідає і в якому сценарії працюємо, а також бізнес-контекст(наприклад, статус замовлення чи дані з CRM). Вся ця інформація на рівні платформи розосереджена між кількома сутностями, але для моделі ШІ її потрібно подати як єдину, лінійну послідовність тексту.

При цьому виникають два фундаментальні обмеження. Перше – це обмеження розміру контексту, притаманне будь-якій великій мовній моделі. Кількість токенів, яку модель може опрацювати за один виклик, є скінченною. Це означає, що історія взаємодії, яка зберігається в базі даних за місяці або роки, не може бути повністю

передана в кожному запиті. Платформа змушена вибирати: які саме фрагменти діалогу є релевантними для поточної відповіді, а які можна відсікати або стискати.

Друге обмеження пов'язане з фрагментацією даних у багатоплатформеному середовищі. Один і той самий користувач може взаємодіяти з ботом через різні канали: наприклад, почати діалог у веб-чаті, продовжити його в месенджері, а потім повернутися до веб-інтерфейсу. Формально всі ці події представлені уніфікованою моделлю Message, але з погляду моделі ШІ вони знову розсипаються на окремі шматки тексту, які потрібно зібрати в єдиний контекст. Якщо платформа не враховує багатоканальність, модель може або не бачити частину важливих попередніх реплік, або отримувати надмірні, неструктуровані дані, що погіршують якість відповіді й наближають нас до межі контексту [36].

На концептуальному рівні процес формування prompt'у можна описати як алгоритм відбору й конструювання контексту на основі історії Message для конкретної пари «бот – контакт». Нижче наведено узагальнений алгоритм, який відображає базову логіку такої побудови.

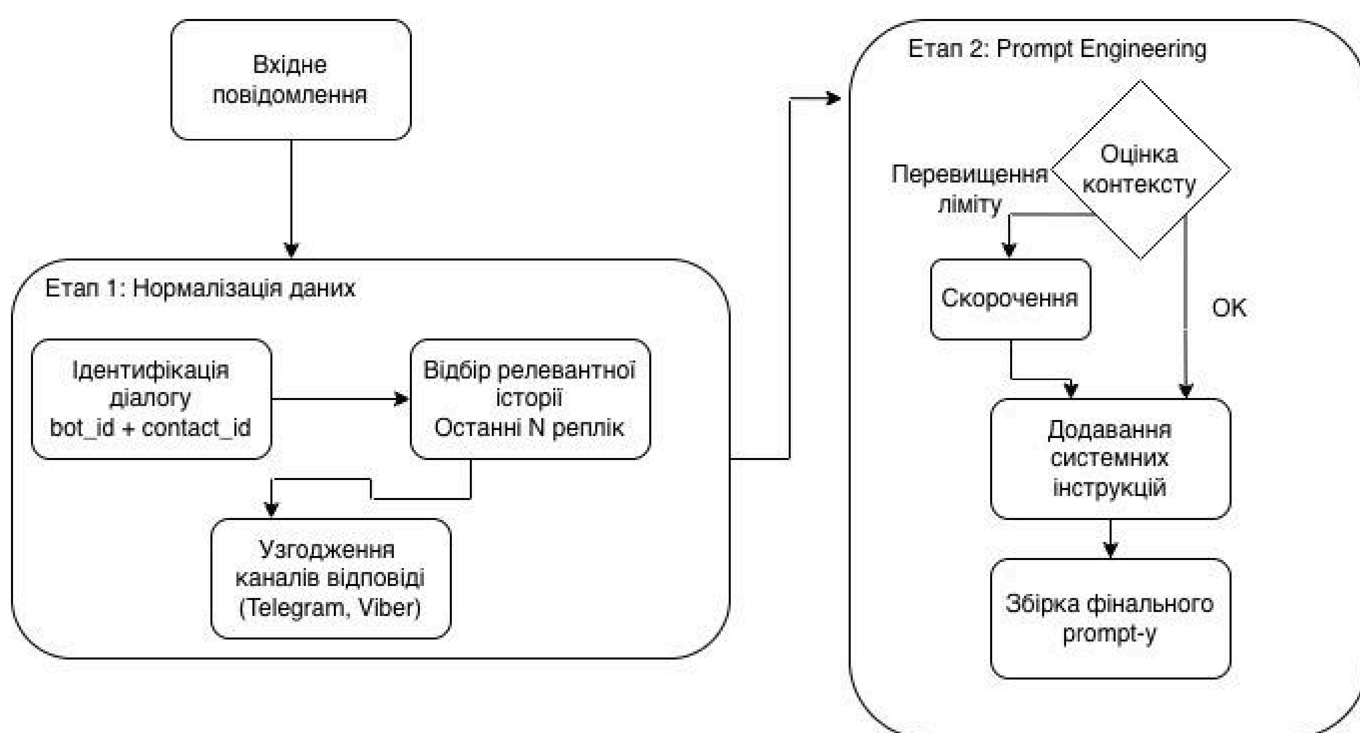

Рис. 3.11. Алгоритм формування контексту та prompt-у для виклику мовної моделі

Такий алгоритм не залежить від конкретного постачальника моделей чи способу розгортання(локально або через API) і безпосередньо спирається на уніфіковану модель повідомлень, описану в попередньому підрозділі. Він показує, що якість інтеграції ШІ визначається не стільки розумністю самої моделі, скільки здатністю платформи коректно відібрати, узгодити та компактно подати контекст у межах доступного вікна контексту. Це створює основу для подальшого аналізу вибору конкретних моделей і методів їхньої інтеграції в умовах розроблюваної SaaS-платформи.

3.4.3 Інтеграція LLM у подієвий конвеєр обробки повідомлень

Інтеграція LLM у синхронну схему обробки повідомлень призводить до серйозних обмежень масштабованості, оскільки виклик моделі може бути тривалим і залежним від зовнішніх ресурсів. Саме тому у межах розроблюваної платформи LLM вбудовується у подієву архітектуру як окремий асинхронний етап.

Після надходження нового повідомлення та його збереження у вигляді об’єкта Message платформа ініціює окрему подію, що відповідає за інтелектуальну обробку. Формування контексту, виклик LLM та отримання результату виконуються незалежно від приймання нових повідомлень. Отримана відповідь повертається у систему у вигляді окремої події, яка також фіксується в історії діалогу та проходить стандартний шлях доставки у відповідний канал.

Такий підхід дозволяє розв’язати кілька проблем одночасно. По-перше, платформа не блокує приймання нових подій під час тривалих викликів моделі. По-друге, з’являється можливість керувати чергами запитів до LLM, обмежувати паралелізм та реалізовувати повторні спроби у разі помилок. По-третє, інтелектуальна обробка стає прозорою з точки зору моніторингу, оскільки кожен виклик моделі представлений як окрема подія з чітким життєвим циклом.

На рисунку 3.12 подано узагальнену схему включення великої мовної моделі в подієвий конвеєр обробки повідомлень, побудований навколо уніфікованої моделі Message. Логіка процесу починається з надходження події з зовнішнього каналу, після чого виконується її нормалізація у внутрішню модель Message. Такий крок

дозволяє одразу відокремити специфіку конкретної платформи від подальшої бізнес-логіки, оскільки всі наступні операції працюють із єдиним форматом даних.

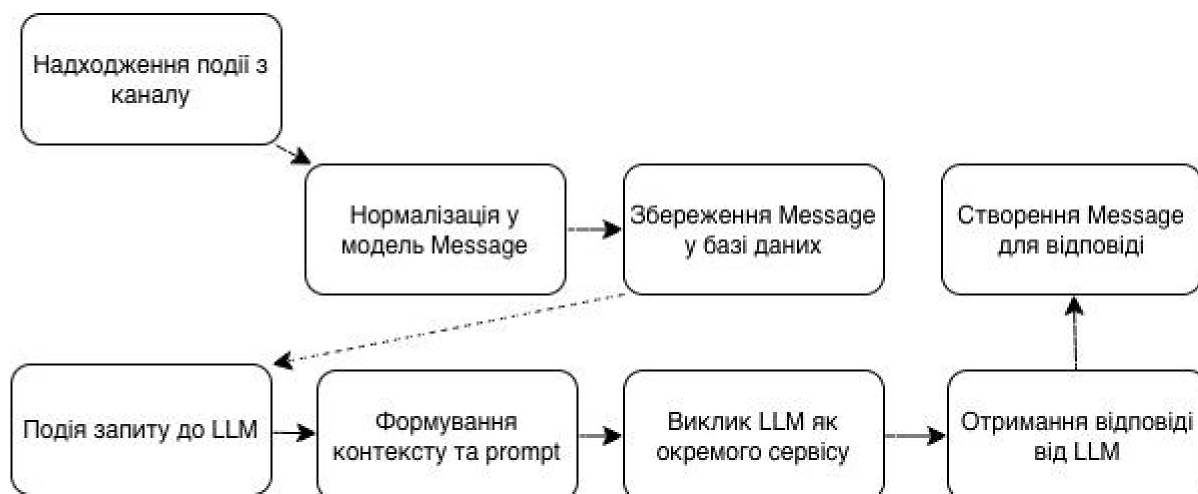

Рис. 3.12. Включення LLM у подієвий конвеєр Message

Після нормалізації повідомлення фіксується в базі даних, що забезпечує збереження історії діалогу незалежно від подальшого сценарію обробки. На цьому етапі конвеєр розгалужується: з одного боку, платформа має запис Message як факт події, з іншого – ініціюється подія запиту до LLM, яка запускає інтелектуальну обробку. Саме ця подієва ініціація є ключовою відмінністю від синхронних схем, оскільки виклик моделі не блокує приймання нових повідомлень і не прив'язаний до часових обмежень транспортного протоколу.

Далі система формує контекст і prompt на основі збережених даних: історії повідомлень, метаданих, інформації про контакт і сценарій. Сформований запит передається до LLM як до окремого сервісу, після чого результат повертається у вигляді відповіді моделі. На схемі підкреслено, що відповідь LLM не "перезаписує" вхідне повідомлення, а використовується для створення нового об'єкта Message, який представляє репліку бота та може бути збережений і оброблений так само, як будь-яке інше повідомлення в системі.

Таким чином, рисунок демонструє, що інтеграція LLM реалізується як окремий етап подієвого конвеєра, де кожен кро –, нормалізація, збереження, формування контексту, виклик моделі та створення відповіді – є відокремленим і

керованим. Це дозволяє масштабувати інтелектуальну обробку незалежно від приймання подій, контролювати навантаження на модель через черги, а також забезпечувати прозорість життєвого циклу кожного запиту до LLM через фіксацію всіх проміжних результатів у структурі Message.

3.4.4 Підходи до розгортання та використання мовних моделей

З архітектурної точки зору інтеграція LLM може здійснюватися двома основними способами: шляхом локального розгортання моделей у власній інфраструктурі або шляхом використання моделей як зовнішнього сервісу через API. Локальний підхід забезпечує повний контроль над даними, затримками та політиками безпеки, що є критичним для систем, які працюють з чутливою інформацією. Водночас він потребує значних обчислювальних ресурсів і постійної підтримки моделей.

Сервісний підхід, навпаки, дозволяє швидко інтегрувати сучасні мовні моделі без необхідності розгортання власного кластера. Це особливо привабливо для SaaS-платформ, де навантаження може бути нерівномірним, а еластичне масштабування є критичним. Проте такий варіант створює залежність від зовнішнього провайдера, додає мережеві затримки та вимагає окремого аналізу питань конфіденційності та вартості.

Незалежно від обраного підходу ключовим залишається принцип ізоляції інтелектуального компонента від основної бізнес-логіки. LLM має бути інтегрована як керований сервіс у межах подієвої архітектури, а всі дані, які вона отримує, повинні формуватися на основі уніфікованої моделі Message. Це забезпечує узгодженість історії діалогу, контроль поведінки системи та можливість еволюційного розвитку платформи без прив’язки до конкретної реалізації мовної моделі.

Висновок до розділу

У цьому розділі виконано проєктування важливих моделей, методів і алгоритмів інтеграції чат-ботів у платформу з урахуванням проблем, виявлених у

попередньому розділі, насамперед фрагментації підключень і слабкої керованості потоку подій у багатоканальному середовищі. Вихідною позицією стало те, що перед побудовою власного інтеграційного шару необхідно чітко розуміти типові практичні моделі підключення ботів до платформ обміну повідомленнями, оскільки саме ці моделі визначають характер подій, вимоги до інфраструктури та межі масштабування.

Було розглянуто базові моделі доставки подій, зокрема підходи з періодичним опитуванням та використанням веб-хуків, і показано, що кожен із них має внутрішні обмеження при масштабуванні. Запропонована модель організації черг на основі RabbitMQ дозволяє відокремити приймання повідомлень від їх обробки, забезпечити асинхронність, керовану маршрутизацію та стійкість до пікових навантажень. Запропонована подієва схема обробки з брокером повідомлень забезпечує кращу масштабованість і спостережуваність у порівнянні з лінійною синхронною обробкою, оскільки дозволяє незалежно масштабувати споживачів, згладжувати пікові навантаження та підвищувати надійність доставки подій.

Окрему увагу приділено інтеграції великих мовних моделей як інтелектуального компонента системи. Показано, що LLM у межах розроблюваної платформи має розглядатися не як автономний агент, а як сервісний етап подієвого конвеєра, який працює з контекстом, сформованим платформою на основі уніфікованої моделі Message. Інтеграція LLM як керованого сервісного етапу конвеєра на основі уніфікованої моделі Message підвищує стабільність та якість відповідей у багатоканальному середовищі завдяки централізованому формуванню контексту, контролю навантаження та збереженню цілісної історії діалогу.

РОЗДІЛ 4
ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Архітектура розробленої SaaS-платформи

4.1.1 Клієнт-серверна модель

Розроблена платформа побудована за клієнт–серверною архітектурою, у якій користувач взаємодіє із системою через веб-інтерфейс, а вся прикладна логіка та інтеграційні сценарії виконуються на серверній стороні. Такий підхід дозволяє ізолювати складні операції від клієнтської частини, яка відповідає переважно за представлення даних та ініціювання команд керування.

Серверна частина реалізована мовою програмування Python із використанням фреймворку FastAPI, який забезпечує зручну побудову HTTP API, підтримку асинхронної обробки запитів та декларативне описання контрактів даних. Використання FastAPI є практично виправданим для систем, де одночасно відбувається велика кількість звернень від клієнта, а також є необхідність швидко реалізовувати інтеграційні точки для зовнішніх каналів комунікації. Зокрема, API платформи виступає центральним вузлом, через який клієнт виконує дії з конфігурації та керування ботами, а також отримує інформацію про їх стан і результати роботи [38].

Показовою є те, що навіть стартова конфігурація застосунку виглядає компактно та читається як опис життєвого циклу сервісу: визначається контекст запуску, задаються метадані, реєструються проміжні обробники запитів(middleware) та підключається маршрутизація. У результаті отримуємо сервіс, який готовий приймати запити клієнта і виступати центральною точкою керування всіма функціями платформи, що можна побачити в лістингу 4.1

Лістинг 4.1

```
logging.basicConfig(level=logging.INFO)

@asynccontextmanager
```

```
async def lifespan(app: FastAPI):
    """Lifespan context manager for the FastAPI application."""
    yield

app = FastAPI(
    title=settings.app_name,
    version=settings.version,
    openapi_url=settings.openapi_url,
    docs_url=settings.docs_url,
    lifespan=lifespan,
)

register_middlewares(app)
app.include_router(router)

if __name__ == "__main__":
    uvicorn.run(
        "app.main:app",
        host=settings.api_host,
        port=settings.api_port,
        reload=settings.app_reload,
        factory=False,
    )
```

Наведений фрагмент демонструє один з плюсів використання FastAPI як легкого фреймворку: для старту системи достатньо описати конфігурацію застосунку, підключити маршрути та запустити ASGI-сервер. Водночас така простота старту не обмежує подальший розвиток, а навпаки, вона забезпечує чисту базу для нарощування логіки обробки подій, підключення інтеграцій, реалізації механізмів контролю доступу та логування за допомогою інтеграції необхідних модулів. Таким чином, використання FastAPI у серверній частині підтримує вимоги щодо централізації складних процесів, керованості та передбачуваної роботи платформи в багатокористувацькому режимі.

Клієнтська частина реалізована як веб-застосунок на базі React.js, що дозволяє будувати інтерфейс як набір повторно використовуваних компонентів та

підтримувати динамічне оновлення станів без надмірних перезавантажень сторінки. Для стилізації використовується Tailwind CSS, що забезпечує уніфікований дизайн і пришвидшує розробку інтерфейсу завдяки набору готових утилітарних класів. У контексті SaaS-платформи це важливо, оскільки користувач працює з інтерфейсом регулярно, а читабельність і структурованість екранів керування напряму впливають на ефективність адміністрування ботів [39].

У межах клієнт-серверної взаємодії критичною вимогою є стабільність та однозначність обміну даними. Для цього сервер надає стандартизовані відповіді у форматі JSON, а клієнт використовує API як єдине джерело істини щодо станів ботів, історії повідомлень та результатів виконання команд. Таким чином, веб-інтерфейс не містить бізнес-логіки обробки повідомлень, а виступає інструментом керування.

4.1.2 Технологічний стек платформи та роль основних компонентів

Для зберігання даних використовується PostgreSQL як реляційна база даних. Її застосування обґрунтовується необхідністю зберігати структуровані сутності платформи (користувачі, конфігурації ботів, інтеграційні налаштування, історія повідомлень, журнали подій, технічні стани) із забезпеченням цілісності, транзакційності та можливості формувати вибірки для відображення у веб-інтерфейсі. Реляційна модель зручна для SaaS-сценарію, оскільки природно підтримує розмежування даних між користувачами та формує основу для контролю доступу.

Окремою практично важливою особливістю зберігання даних є використання власної конфігурації на сервері, яка дозволяє централізовано керувати параметрами інфраструктурних компонентів(зокрема підключенням до бази даних) без жорсткого хард-кодингу значень у коді. Такий підхід повинен коректно працювати в різних середовищах(локальна розробка, тестування, продакшн), а відмінності між ними мають контролюватися через конфігурацію.

З практичної точки зору конфігурація застосунку включає не лише сам рядок підключення до PostgreSQL, а й параметри пулу з’єднань. Такі параметри напряму

впливають на стабільність та продуктивність системи в багатокористувацькому режимі, оскільки навантаження на базу даних може зростати зі збільшенням кількості активних ботів і обсягу подій [40]. Використання асинхронного ORM SQLAlchemy дозволяє ефективно працювати з великою кількістю паралельних запитів, не блокуючи основний цикл обробки подій у сервері, а з кодом цієї конфігурації можна ознайомитись в лістингу 4.2.

Лістинг 4.2

```
DATABASE_URL = db_settings.database_url

engine = create_async_engine(
    DATABASE_URL,
    pool_size=db_settings.database_pool_size,
    max_overflow=db_settings.database_max_overflow,
    echo=False,
    future=True,
)
AsyncSessionLocal = sessionmaker(engine, class_=AsyncSession,
expire_on_commit=False)

async def get_db() -> AsyncGenerator[AsyncSession, None]:
    async with AsyncSessionLocal() as session:
        try:
            yield session
            await session.commit()
        except Exception:
            await session.rollback()
            raise
        finally:
            await session.close()

async def get_standalone_database():
    return AsyncSessionLocal()

@asynccontextmanager
async def get_context_db():
```

```
    async with AsyncSessionLocal() as session:
        try:
            yield session
            await session.commit()
        except Exception:
            await session.rollback()
            raise
        finally:
            await session.close()
```

Наведений фрагмент конфігурації системи показує, що доступ до бази даних реалізовано як уніфікований сервісний механізм, параметризований через власний конфіг.

Поряд із конфігурацією доступу до PostgreSQL у платформі реалізовано окремий інфраструктурний шар для подійної взаємодії через RabbitMQ. Для SaaS-системи це принципово важливо, оскільки дозволяє рознести у часі приймання запитів через HTTP API та виконання ресурсоємних операцій, наприклад обробку повідомлень, журналювання подій, виконання інтеграційних запитів або формування відповіді. Відповідно, серверна частина зберігає стабільну швидкодію, а фонові компоненти можуть обробляти події асинхронно, не блокуючи користувацькі сценарії.

Так само як і для бази даних, параметри RabbitMQ винесено у власну конфігурацію. Адреса підключення та ліміти пулів, наприклад кількість одночасних з’єднань і каналів, задаються централізовано. Це спрощує експлуатацію та масштабування, оскільки при зміні навантаження або переході між середовищами, наприклад локальним, тестовим або продуктивним, немає потреби змінювати прикладний код, оскільки достатньо адаптувати конфігураційні значення. Додатковою перевагою є використання пулів: повторне використання з’єднань і каналів зменшує накладні витрати на створення нових підключень, що особливо відчутно при великій кількості подій і паралельній роботі воркерів. А з цією конфігурацією можна ознайомитись на лістингу 4.3.

Лістинг 4.3.

```
logger = logging.getLogger(__name__)

async def get_connection():
    return await aio_pika.connect_robust(rabbitmq_settings.rabbitmq_url)

connection_pool = Pool(
    get_connection,
    max_size=rabbitmq_settings.rabbitmq_connection_pool_size
)

async def get_channel():
    async with connection_pool.acquire() as connection:
        return await connection.channel()

channel_pool = Pool(
    get_channel,
    max_size=rabbitmq_settings.rabbitmq_channel_pool_size
)

async def get_rabbitmq_client():
    async with channel_pool.acquire() as channel:
        yield channel

async def get_standalone_rabbitmq_client():
    return await get_channel()
```

У наведеному фрагменті реалізовано два режими доступу до RabbitMQ. Перший режим орієнтований на контрольований доступ через генератор, який повертає готовий канал із пулу. Другий режим підходить для окремих службових сценаріїв, наприклад коли потрібен прямий доступ до каналу без використання залежностей або контексту виконання. Таким чином, брокер повідомлень підключається не як допоміжний елемент, а як повноцінний інфраструктурний компонент із централізованим налаштуванням та оптимізованим використанням ресурсів. У контексті розділу 3 це підтримує рішення, спрямовані на підвищення керованості та стабільності системи під навантаженням: подійна обробка не

перевантажує API, а доступ до брокера організовано передбачувано та масштабовано.

Оскільки платформа поєднує інтерактивний керуючий інтерфейс і подійну обробку повідомлень, кожен компонент стека виконує чітку роль: клієнтська частина забезпечує зручну взаємодію між такими компонентами як API, доступ до функцій бізнес-логіки і контроль, роботи з базою даних та збереження стану та історії. У сукупності це формує архітектурний каркас, який дозволяє реалізувати SaaS-підхід як єдиний сервіс, доступний через веб.

4.1.3 Обґрунтування вибору SaaS-підходу та його роль у реалізації системи

Вибір моделі SaaS обумовлений необхідністю забезпечити централізований доступ до функціональності платформи для різних користувачів та різних сценаріїв використання, без залежності від локального середовища клієнта. На практиці SaaS-підхід дозволяє зосередити ключові механізми – керування ботами, налаштування інтеграцій, обробку подій і моніторинг – у єдиному контрольованому середовищі, що підвищує надійність та спрощує супровід.

З позиції експлуатації SaaS-модель забезпечує оперативне оновлення функціональності та уніфіковані правила роботи для всіх користувачів. Це важливо для системи чат-ботів, оскільки інтеграції з месенджерами, формат подій та вимоги до безпеки змінюються, а платформа повинна швидко адаптуватися. У такому підході користувач отримує доступ до керування та спостереження через веб-інтерфейс, а виконання складних сценаріїв відбувається на серверному боці, де є можливість гарантувати контроль ресурсів, журналювання та перевірку станів.

Окремо слід підкреслити, що SaaS-модель напряму підтримує реалізацію рішень, згаданих в попередньому розділі. Зокрема, централізація керування та стандартизація процесів дозволяють впровадити єдині правила обробки повідомлень, контроль асинхронних задач, а також уніфікований підхід до інтеграції мовної моделі, де якість відповідей залежить від коректності контексту та правил взаємодії.

4.2 Реалізація модулів управління ботами

4.2.1 Підтримувані платформи та єдина модель виконання ботів

На поточному етапі платформа підтримує створення та запуск ботів для двох платформ: Telegram і Viber. Незважаючи на різницю протоколів, обидва типи ботів реалізовані за єдиною концепцією виконання. Кожен бот запускається як окремий керований процес у контейнеризованому середовищі, а обмін подіями з ядром платформи здійснюється через брокер повідомлень.

Такий підхід дозволяє уніфікувати обробку подій. Незалежно від того, з якого каналу надійшло повідомлення, бот формує структурований payload і передає його в RabbitMQ. Далі повідомлення обробляється серверними компонентами платформи, що відокремлює приймання подій від основної логіки та підвищує стабільність.

4.2.2 Реалізація runner-компонента на прикладі Telegram-бота

Runner-компонент відповідає за підключення до конкретного каналу та перетворення вхідних подій у внутрішній формат платформи. На прикладі Telegram-бота ключовими є такі можливості: ініціалізація runner-а з ідентифікатором бота, формування payload та публікація повідомлення в RabbitMQ.

Лістинг 4.4.

```
class TelegramBotRunner(BaseBot):
    """Runner for Telegram bot that handles message routing via
RabbitMQ."""

    def __init__(self, bot_id: int, api_key: str) -> None:
        self.bot_id = bot_id
        self.api_key = api_key
        self.app = None
        self.redis = None
```

Наведений фрагмент демонструє, що runner зберігає мінімально необхідні параметри, а саме ідентифікатор бота та ключ доступу. Це дозволяє однозначно

прив’язати всі події до конкретного екземпляра в системі та коректно маршрутизувати обробку повідомлень у межах платформи.

Лістинг 4.5

```
async def _publish_to_rabbitmq(self, exchange_name: str, routing_key: str,
payload: dict) -> None:
    channel = await get_standalone_rabbitmq_client()
    exchange = await channel.declare_exchange(
        exchange_name,
        type=ExchangeType.DIRECT,
        durable=True,
    )

    message = Message(
        body=json.dumps(payload).encode(),
        delivery_mode=DeliveryMode.PERSISTENT,
    )

    await exchange.publish(message, routing_key=routing_key)
```

Цей метод реалізує перетворення внутрішнього об’єкта події у формат, придатний для передачі брокеру повідомлень. Платформа використовує прямий тип маршрутизації через exchange і routing key, що дозволяє відокремити різні види подій та, за потреби, масштабувати обробку окремими групами воркерів. Використання персистентної доставки підсилює надійність у сценаріях, коли повідомлення повинні бути збережені в брокері до моменту обробки. Нижче представлено лістинг 4.6, на якому бачно формування payload із прив’язкою до бота та передавання сирих подій.

Лістинг 4.6

```
async def send_rabbitmq_message(self, update: Update) -> None:
    payload = {
        "bot_id": self.bot_id,
        "is_from_bot": False,
```

```
        "raw": update.to_dict(),
    }

    await self._publish_to_rabbitmq(
        bot_settings.bot_message_incoming_exchange,
        bot_settings.bot_message_incoming_telegram_routing_key,
        payload,
    )
```

Payload містить три основні елементи. Поле bot_id ідентифікує конкретний бот у системі. Поле is_from_bot використовується для розмежування подій за джерелом, наприклад вхідні повідомлення користувача та вихідні повідомлення бота. Поле raw зберігає сирі дані події, що дозволяє відтворити первинний контекст під час аналізу або журналювання, а також спростити підтримку у випадках зміни формату подій з боку зовнішньої платформи. Що ж до фільтрації на вході, то з базовою функцією для цього можна ознайомитись на лістингу 4.7.

Лістинг 4.7.

```
async def handle_message(self, update: Update, context) -> None:
    if not update.message or not update.message.text:
        return

    user = update.effective_user
    text = update.message.text
```

У цьому фрагменті показано мінімальну валідацію вхідних даних. Runner відкидає нерелевантні події та залишає лише ті, що мають текстове повідомлення. Це зменшує зайве навантаження на подальші компоненти й забезпечує стабільний вхідний формат для маршрутизації в RabbitMQ.

Підхід із runner-компонентом дозволяє аналогічно реалізувати Viber-бота. Різниця полягатиме в механізмі отримання подій від каналу, проте частина з формуванням payload і публікацією в RabbitMQ залишається уніфікованою.

4.2.3 Контейнеризація ботів і запуск через Python Docker SDK

Оскільки платформа функціонує в контейнеризованому середовищі, кожен бот запускається як окремий контейнер з відповідним runner-кодом. Такий підхід забезпечує ізоляцію виконання, керованість життєвого циклу та можливість паралельної роботи кількох ботів. Запуск і зупинка контейнерів здійснюється програмно за допомогою Python Docker SDK, що дозволяє реалізувати керування ботами як частину бізнес-логіки платформи.

У рамках модулів управління ботами реалізуються типові операції, наприклад створення контейнера, запуск, зупинка, перезапуск, а також передавання необхідних параметрів, таких як ключ доступу і конфігураційні змінні оточення. Це є прямою реалізацією ідеї керованості, що бот не є "випадковим скриптом", а є контрольованим виконуваним модулем платформи.

Для запуску Telegram-бота використовується окремий Docker-образ, який містить runtime-середовище Python та необхідні залежності. У межах збірки копіюються спільні модулі, runner-компоненти та конфігурації, після чого встановлюються залежності через Poetry. Це дозволяє підтримувати керовані та відтворювані збірки, а також ізолювати залежності різних типів ботів. З вмістом цього Docker-образу можна ознайомитись в лістингу 4.8.

Лістинг 4.8

```
FROM python:3.12-slim AS base
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV PYTHONPATH=/src
RUN apt-get update && apt-get install --no-install-recommends -y
build-essential libpq-dev && rm -rf /var/lib/apt/lists/*
WORKDIR /src
COPY ./shared ./shared
COPY ./runners ./runners
COPY ./migrations ./migrations
COPY ./pyproject.toml ./pyproject.toml
COPY ./poetry.lock ./poetry.lock
COPY ./alembic.ini ./alembic.ini
```

```
RUN pip install poetry
RUN poetry install --only=telegram --no-root --no-ansi --no-interaction

EXPOSE 8000
```

Цей лістинг демонструє, що runner-частина та спільні модулі пакуються в окремий образ, а залежності встановлюються згідно з визначеною групою. Підхід із розділенням залежностей дозволяє уникнути надлишкових бібліотек у контейнері та пришвидшує збірку і доставку. Також це спрощує масштабування, оскільки запуск нового екземпляра бота зводиться до створення та старту контейнера з уже підготовленим образом.

На рівні доменної моделі кожен бот описується не лише ідентифікатором і API-ключем, а й типом провайдера, що дає змогу service-ядру вибрати відповідний образ і команду запуску. Логіка керування контейнерами може бути викликана як із адміністративного веб-інтерфейсу (підключення/відключення бота користувачем SaaS-платформи), так і з внутрішніх сервісів, які реалізують автоматичний перезапуск чи масштабування. Візуальним підтвердженням описаної архітектури є стан контейнерів під час запуску системи, зафіксований на рисунку 3.1, який взятий з Docker Desktop.

Рис 4.1. Стек контейнерів платформи чат-ботів у середовищі Docker Desktop

На рисунку 4.1 показано, що всі компоненти платформи згруповано в один стек chat-bot-system, до якого входять окремі контейнери бази даних db-1, брокера повідомлень rabbitmq-1, кеша redis-1, сервісу Docker-in-Docker dind-1, а також прикладних сервісів backend-1, observer-1, consumer-1 та frontend-1. Власне Docker Engine позначено як окремий верхньорівневий елемент, тоді як стек платформи відображається як логічно пов'язаний набір контейнерів, що взаємодіють між собою в межах спільної мережі.

4.2.4 Керування життєвим циклом бота через веб-інтерфейс

Оскільки платформа реалізована як SaaS-рішення, керування життєвим циклом ботів виконується централізовано через веб-додаток. Це означає, що користувач працює не з "локальним запуском" скриптів, а з керованими екземплярами ботів, для яких система забезпечує створення, запуск, контроль стану та подальше адміністрування. Такий підхід відповідає логіці, описаній у розділі 3 у межах вирішення проблем, де пріоритетом є керованість, передбачуваність роботи та контроль станів при експлуатації. На рисунку 4.2 показано сторінку керування ботами у веб-інтерфейсі.

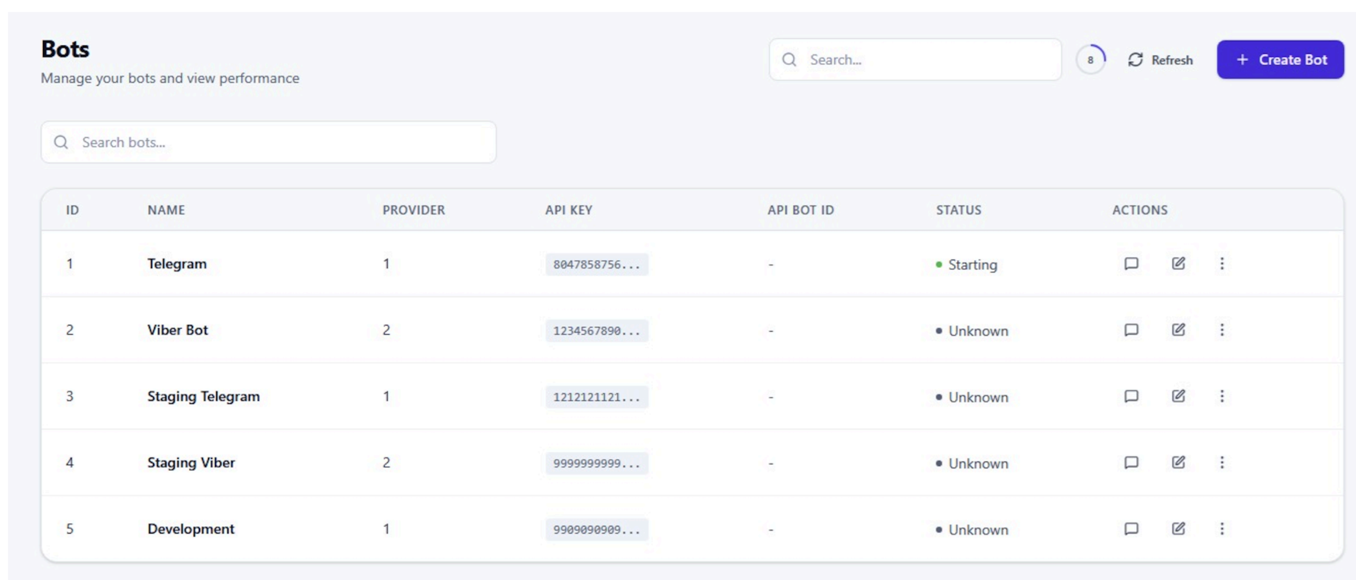

Рис. 4.2. Сторінка керування чатботами

Важливо, що інтерфейс не лише демонструє стан, але й фактично виступає

точкою керування. Користувач може ініціювати створення нового бота, що буде розглянуто в наступних підрозділах.

4.3 Реалізація AI-компонента

4.3.1 Сервіс генерації підказок і отримання LLM-клієнта

AI-компонент платформи реалізовано як окремий сервіс прикладного рівня, який відповідає за формування підказок відповідей у чатах та виконання довільних запитів до мовної моделі. Важливо, що система не прив’язується до одного постачальника. Користувач може обрати хмарний провайдер, наприклад OpenAI, Anthropic або Google, або використати локальну модель, наприклад через Ollama чи LM Studio. Такий підхід відповідає рішенню з розділу 3, де наголошувалося на необхідності гнучкої інтеграції LLM без жорсткої залежності від конкретного сервісу.

Логіка AI-функцій зосереджена у сервісі, який працює від імені авторизованого користувача та використовує його налаштування. Ключова ідея полягає в тому, що сервер не зберігає “вшиті” параметри моделі, а отримує готовий клієнт через сервіс налаштувань. Якщо користувач не обрав провайдера або не вказав ключ доступу, сервіс повертає зрозумілу помилку і пропонує спочатку виконати конфігурацію. З частиною оголошенням класа цього сервісу можна ознайомитись в лістингу 4.9.

Лістинг 4.9

```
class SuggestionsService:
    def __init__(self, db, user, contacts_service, settings_service):
        self.db = db
        self.user = user
        self.contacts_service = contacts_service
        self.settings_service = settings_service

    async def _get_llm_client(self):
        llm_client = await self.settings_service.get_llm_client()
```

```
        if not llm_client:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="LLM not configured. Please configure your LLM
settings first.",
            )
        return llm_client
```

Цей фрагмент демонструє практичну реалізацію "уніфікованого" підходу. Сервіс працює з абстракцією клієнта, а конкретний провайдер визначається налаштуваннями користувача. Саме завдяки цьому одна і та сама бізнес-логіка може працювати як із провайдерами через API, так і з локальними LLM.

4.3.3 Генерація підказок відповіді та інтеграція з чатами

Після отримання LLM-клієнта та підготовки історії діалогу сервіс викликає метод генерації підказок. На виході повертається набір коротких варіантів відповіді, які відображаються у веб-інтерфейсі як швидкі дії для оператора. Основну функцію для генерації можна побачити в лістингу 4.10

Лістинг 4.10.

```
async def generate_suggestions(self, bot_id: int, contact_id: int,
num_suggestions: int = 3):
    llm_client = await self._get_llm_client()
    conversation_history = await self._get_conversation_history(bot_id,
contact_id)
    suggestions = await llm_client.generate_suggestions(
        conversation_history,
        num_suggestions=num_suggestions,
    )
    if not suggestions:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to generate suggestions",
        )
    return suggestions
```

У цьому фрагменті видно основний потік виконання. Спочатку перевіряється, що LLM налаштовано. Далі формується контекст і виконується запит на генерацію підказок. Якщо відповідь не отримано, система повертає контрольовану помилку, щоб інтерфейс міг коректно відреагувати.

Окрім підказок у чатах, AI-компонент підтримує довільні звернення до моделі. Це корисно для сценаріїв, де потрібно сформувати відповідь за окремим промптом або виконати допоміжну генерацію тексту, наприклад підготувати шаблон повідомлення чи коротке резюме звернення. Реалізацію такого функціоналу можна побачити в лістингу 4.11.

Лістинг 4.11.

```
async def generate_custom_completion(self, prompt: str, system_prompt: 
Optional[str] = None, **kwargs) -> str:
    llm_client = await self._get_llm_client()
    messages = []
    if system_prompt:
        messages.append(LLMMessage(role="system", content=system_prompt))
    messages.append(LLMMessage(role="user", content=prompt))
    response = await llm_client.generate_completion(messages, **kwargs)
    if not response:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to generate completion",
        )
    return response
```

Фрагмент демонструє, що система підтримує формат повідомлень із ролями, що дозволяє застосовувати системні правила і користувацькі запити в одному ланцюгу. Параметри генерації передаються як додаткові аргументи, що спрощує адаптацію під різні моделі та сценарії.

4.3.4 Налаштування LLM через веб-інтерфейс і приклад використання підказок

AI-компонент керовано інтегрований у веб-інтерфейс платформи, що дозволяє користувачу підключати та змінювати LLM без редагування коду. На рисунку 4.3 показано сторінку налаштувань, де користувач обирає провайдера із запропонованого списку. У переліку доступні варіанти хмарних сервісів, а також локальні рішення, що підтверджує можливість використання як API-моделей, так і локальних LLM.

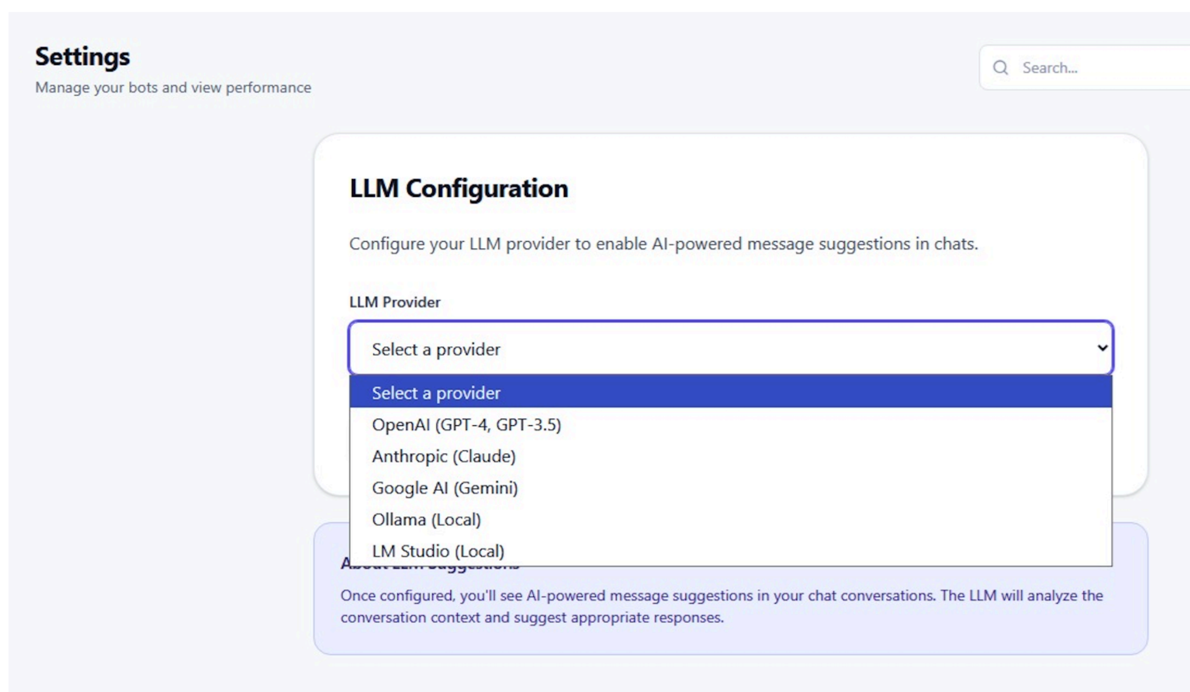

Рис. 4.3. Сторінка налаштувань LLM і вибір провайдера у веб-інтерфейсі

Далі, після вибору провайдера, інтерфейс відображає форму введення параметрів, наприклад ключ доступу та назву моделі. Це показано на рисунку 4.4, де обрано провайдера та вказано параметри для подальшої роботи. Такий підхід уніфікує процес підключення різних LLM-сервісів і спрощує подальшу зміну провайдера без модифікації основної логіки застосунку. Після збереження налаштувань система використовує ці значення для побудови LLM-клієнта на серверній стороні, що узгоджується з логікою сервісу, наведеною в лістингу 4.9.

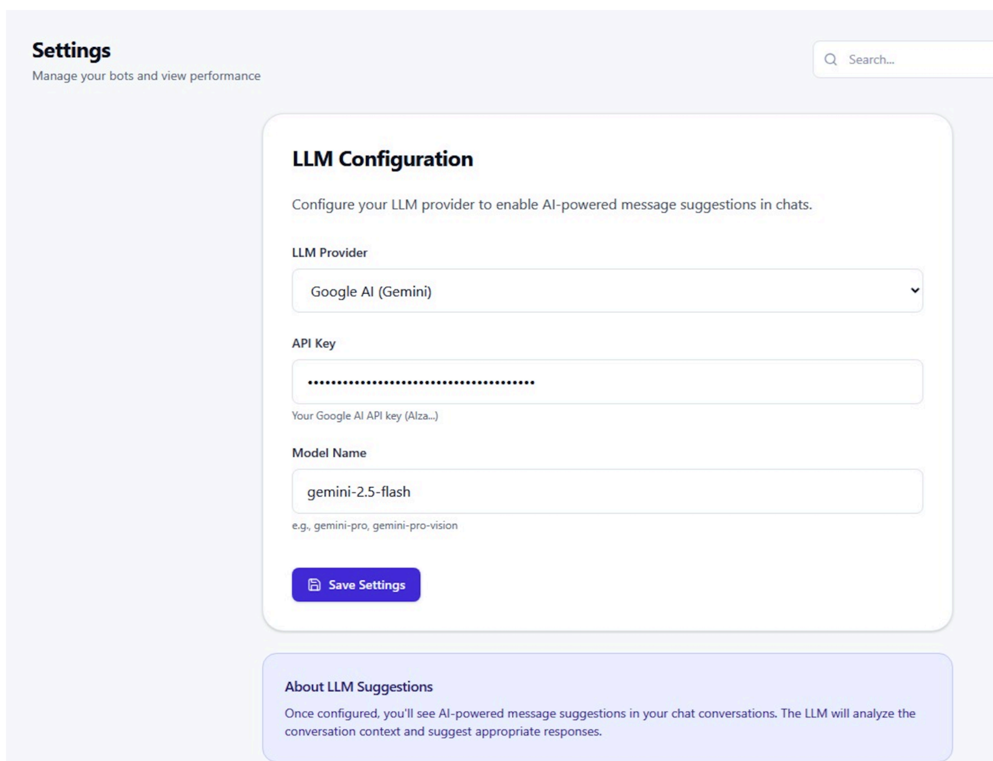

Рис 4.4. Приклад конфігурації провайдера та параметрів моделі

Після конфігурації LLM підказки стають доступними безпосередньо в чаті. На рисунку 4.5 показано інтерфейс діалогу з контактом, у нижній частині якого відображається блок AI Suggestions із кількома варіантами відповіді. Оператор може обрати підготовлений варіант і швидко сформувати реакцію на повідомлення користувача. Таким чином, AI-компонент не замінює керований процес комунікації, а підсилює його, надаючи релевантні підказки в контексті поточного діалогу.

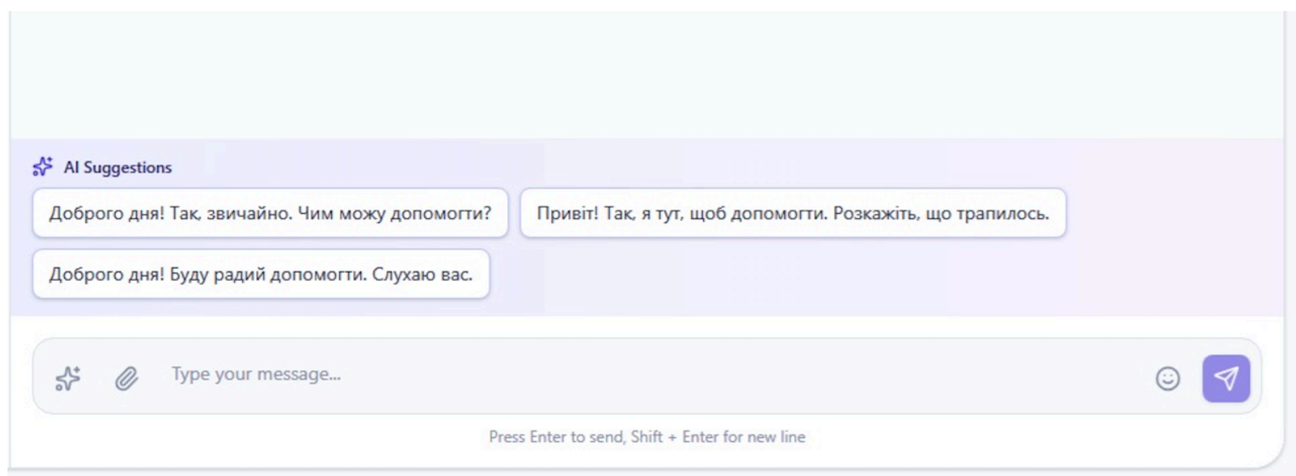

Рис. 4.5. Відображення підказок відповіді в чаті після налаштування LLM

У підсумку, реалізований AI-компонент забезпечує керовану інтеграцію мовних моделей у межах SaaS-платформи та доповнює стандартний процес ведення діалогів інтелектуальними підказками. Таким чином, підхід, продемонстрований у цьому підрозділі, реалізує положення з розділу 3 щодо гнучкого формування контексту та ефективного застосування LLM у задачах підтримки діалогів, забезпечуючи зручність для користувача й масштабованість для подальшого розвитку платформи.

4.4 Демонстрація роботи веб-додатка та користувацьких сценаріїв

4.4.1 Панель моніторингу та створення нового бота

Завершальним етапом реалізації є демонстрація роботи розробленої SaaS-платформи з позиції кінцевого користувача. На рисунку 4.6 представлено головну сторінку у вигляді дашборду. Дашборд відображає зведені показники, які дозволяють швидко оцінити поточний стан системи, зокрема загальну кількість ботів, кількість активних екземплярів та обсяг повідомлень за день. Такий формат подання інформації є зручним для адміністратора, оскільки зменшує час на діагностику та дозволяє оперативно перейти до потрібного функціонального модуля.

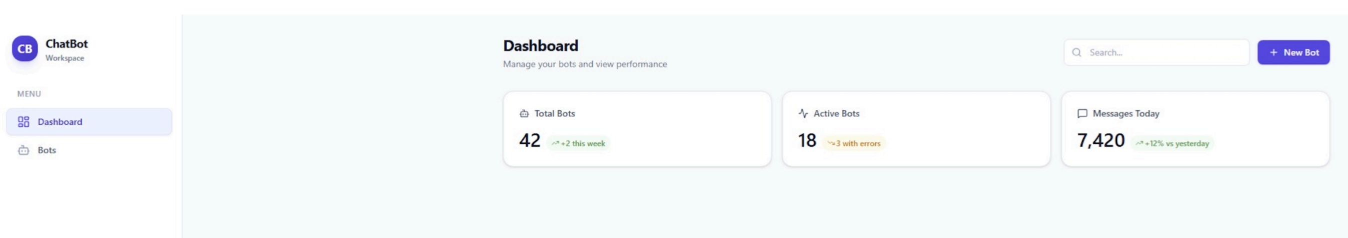

Рис. 4.6. Дашборд із зведеними показниками роботи платформи

Після перегляду зведених показників користувач може ініціювати створення нового бота через відповідну форму. На рисунку 4.7 показано інтерфейс створення, де задаються первинні параметри, необхідні для ініціалізації екземпляра. Користувач задає назву бота, обирає провайдера, наприклад Telegram або Viber, та вводить ключ доступу. На цьому етапі формується базова конфігурація, яка у подальшому

використовується серверною частиною для запуску відповідного runner-компонента та інтеграції із зовнішнім каналом.

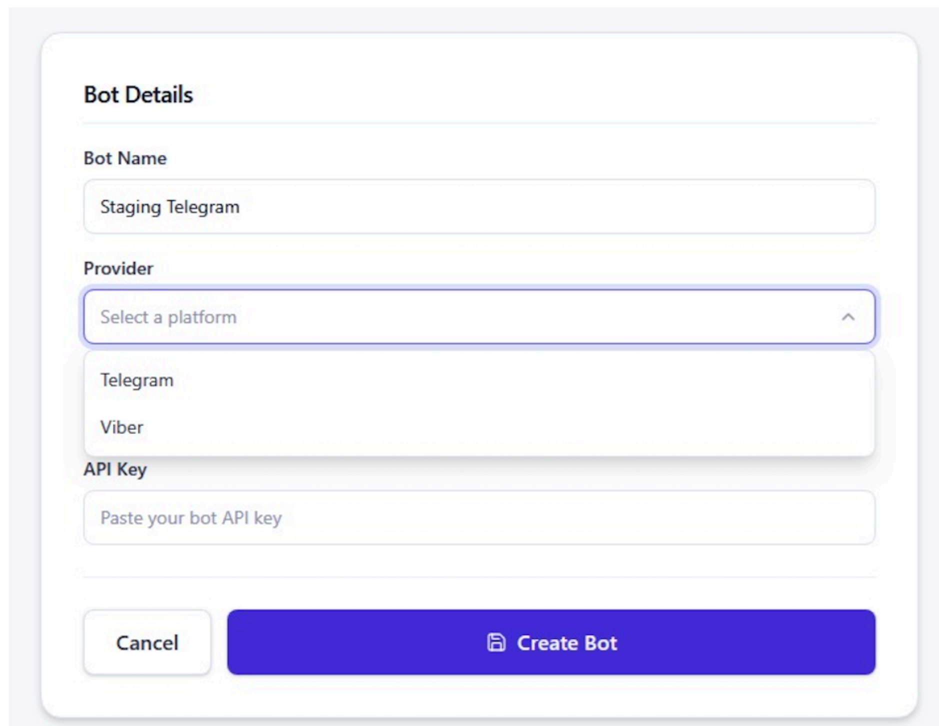

Рис. 4.7. Форма створення бота з вибором платформи

На рисунку 4.7 також наведено заповнення конфігураційного параметра, а саме ключа доступу для обраного провайдера. У практичному сенсі цей ключ забезпечує можливість авторизації в зовнішньому сервісі та отримання подій від користувачів. Після підтвердження створення система додає бота до переліку керованих екземплярів і переводить користувача до подальших сценаріїв роботи, пов'язаних із веденням діалогів та обробкою повідомлень.

4.4.2 Інтерфейс чату, оновлення повідомлень і доставка відповіді в Telegram

Після створення та запуску бота ключовим прикладним сценарієм є ведення комунікації з користувачами через єдиний веб-інтерфейс. На рисунку 4.8 представлено сторінку чату, яка включає список контактів та область перегляду історії повідомлень. Такий підхід дозволяє оператору працювати з діалогами централізовано, не перемикаючись між зовнішніми клієнтами месенджерів. Для

підтримки актуальності історії передбачено механізм оновлення, завдяки якому оператор може синхронізувати вміст чату з фактичними даними, що зберігаються в базі даних платформи.

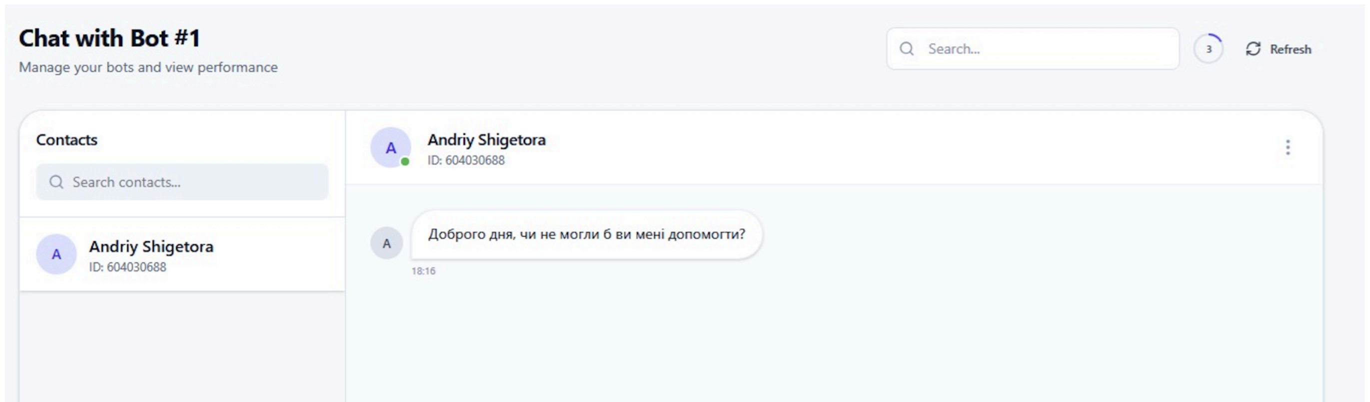

Рис. 4.8. Інтерфейс чату в веб-додатку з історією повідомлень і контактами

Після введення тексту та надсилання повідомлення від імені бота система відображає результат у переписці. Це показано на рисунку 4.9, де видно повідомлення, додане в історію діалогу. На рівні серверної логіки така дія означає фіксацію вихідного повідомлення та його маршрутизацію в зовнішній канал через інтеграційний механізм, описаний у попередніх підрозділах. Таким чином, веб-чат виступає не лише засобом моніторингу, а й інструментом керованої відповіді від імені бота.

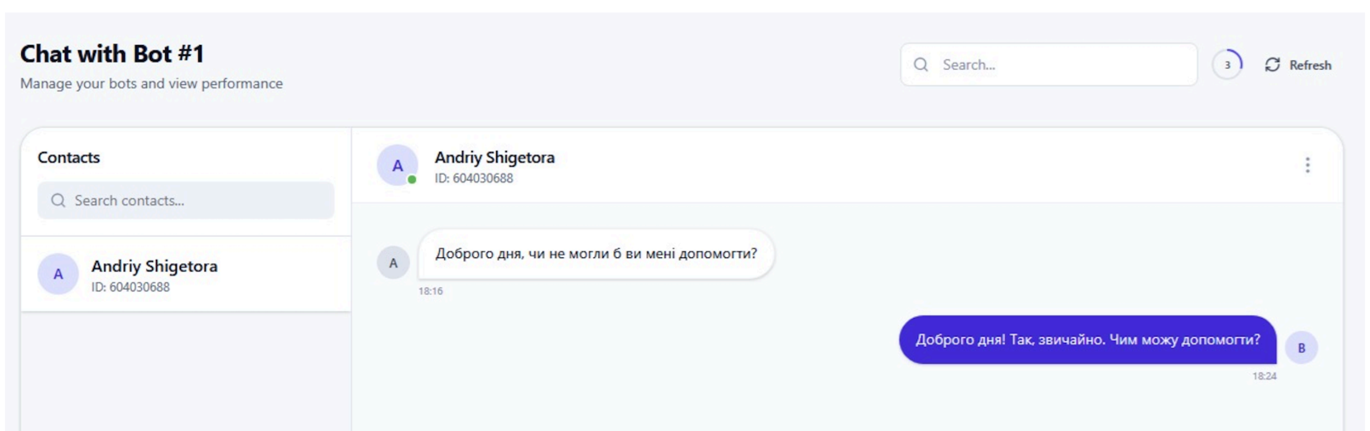

Рис. 4.9. Відображення повідомлення, надісланого з веб-інтерфейсу від імені бота

Факт доставки підтверджується на стороні месенджера. На рисунку 4.U показано, що повідомлення, сформоване у веб-додатку, надходить у Telegram та відображається в діалозі кінцевого користувача.

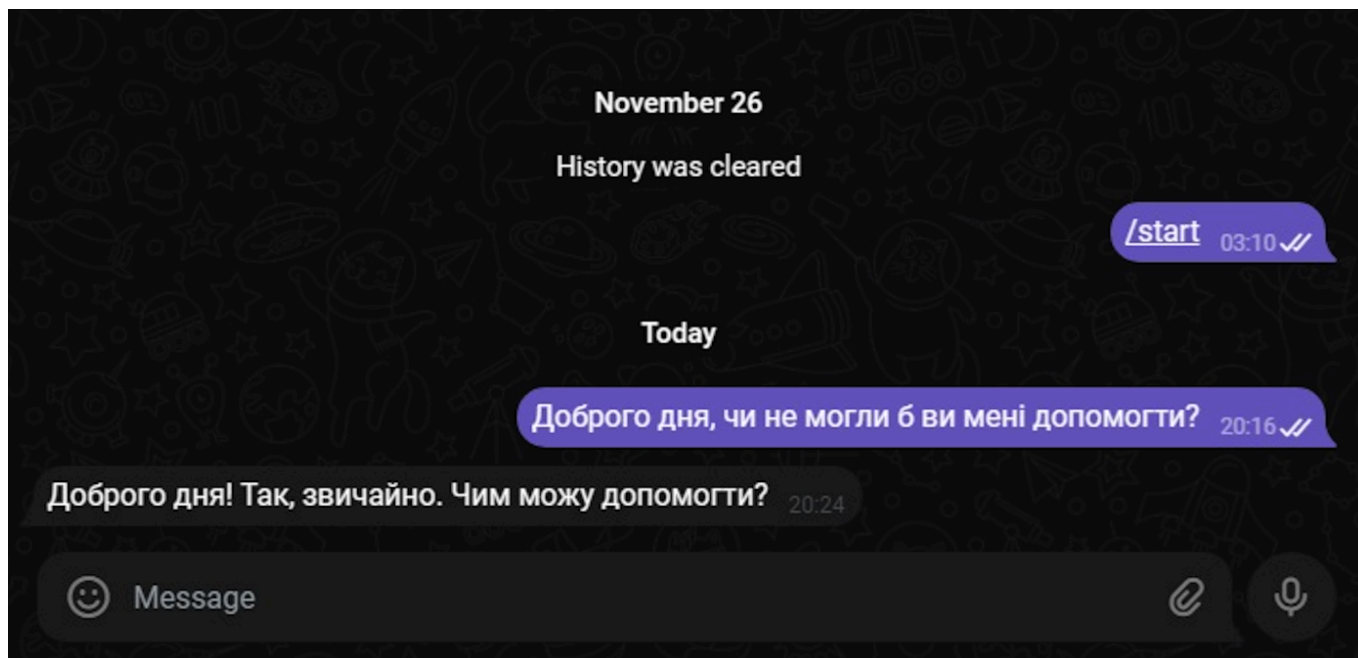

Рис 4.10. Отримання повідомлення в Telegram після надсилання з веб-інтерфейсу

Це демонструє коректність наскрізного сценарію, де платформа виконує роль централізованого середовища для роботи оператора, а користувач отримує відповідь у звичному каналі комунікації. У підсумку така демонстрація підтверджує практичну працездатність системи та узгоджується з положеннями розділу 3 щодо керованості та надійної доставки повідомлень.

Висновок до розділу

У цьому підрозділі було виконано програмну реалізацію та практичну демонстрацію розробленої SaaS-платформи для інтеграції чат-ботів у веб-додаток. Показано, що система реалізована як контейнеризований комплекс взаємопов’язаних сервісів, де прикладні компоненти працюють разом із базовою інфраструктурою. Зокрема, в одному стеку розгортаються база даних, брокер повідомлень, кеш, сервіс

Docker-in-Docker, а також прикладні модулі серверної частини, спостерігача, споживача подій і веб-клієнта, що забезпечує керованість та відокремлення відповідальностей між компонентами.

Окремо реалізовано модулі керування ботами та AI-компонент, які працюють у межах єдиної подієвої схеми: керування життєвим циклом ботів виконується через веб-інтерфейс, що відповідає концепції керованих екземплярів замість локального запуску скриптів. AI-логіка інтегрована як керований сервіс, ізоляційно від основної бізнес-логіки, а дані для обробки формуються на основі уніфікованої моделі повідомлення, що спрощує подальший розвиток і підтримку різних провайдерів LLM. Практична працездатність підтверджена наскрізним сценарієм роботи оператора у веб-чаті: повідомлення, надіслане від імені бота з веб-інтерфейсу, коректно доставляється у месенджер і відображається в діалозі користувача, що підтверджує керованість та надійність доставки, описані в вимогах до цієї системи.

Реалізація платформи у вигляді єдиного контейнеризованого стеку забезпечує відтворюваність розгортання та спрощує масштабування системи порівняно з ручним налаштуванням окремих компонентів і локальним запуском бот-скриптів. У підсумку це зменшує залежність інтеграції від конкретних провайдерів і технічних деталей середовища, підвищуючи швидкість підключення нових ботів/каналів та стабільність роботи в багатокористувацькому режимі.

ВИСНОВКИ

В даній магістерській роботі було розглянуто проблему інтеграції чат-ботів у веб-додатки в умовах багатоплатформеності та різних підходів до організації обміну повідомленнями. Було проаналізовано еволюцію діалогових систем, основні типи чат-ботів та сучасні архітектурні моделі взаємодії, що застосовуються для підключення ботів до месенджерів і веб-сервісів.

У результаті аналізу встановлено, що фрагментація інтерфейсів та відмінності у механізмах отримання подій ускладнюють централізоване управління, масштабування та експлуатаційний контроль, особливо коли кількість ботів і каналів комунікації зростає.

Для вирішення зазначених проблем було обґрунтовано підхід до створення єдиного керованого середовища інтеграції у вигляді SaaS-платформи, яка забезпечує централізований доступ до налаштувань, моніторингу й керування ботами. Додатково наголошено, що уніфікація інтеграції через спільний підхід до подій і даних зменшує залежність від специфіки окремих API та підвищує масштабованість рішення при розширенні кількості каналів і ботів.

Визначено ключові вимоги до такої системи, зокрема ізоляцію виконання ботів, стабільну обробку подій, збереження історії діалогів та можливість розширення на нові платформи без суттєвої перебудови рішення. В межах практичної частини було спроєктовано та реалізовано програмний прототип системи інтеграції чат-ботів, у якому застосовано контейнеризований підхід до розгортання компонентів та використано механізми черг для організації обміну повідомленнями між сервісами. Реалізація продемонструвала можливість централізованого створення та налаштування ботів, перегляду діалогів у веб-інтерфейсі й надсилання повідомлень від імені бота з підтвердженням доставки в месенджері.

Окремо було розглянуто використання великих мовних моделей як інструмента підвищення якості та гнучкості відповідей у діалогових сценаріях. Показано, що застосування LLM дозволяє покращити природність комунікації та

підтримати складніші запити користувачів, а також може бути використане як допоміжний механізм для формування підказок відповідей оператору.

Водночас підкреслено практичний аспект експлуатації: повноцінне донавчання(fine-tuning) потребує значних ресурсів і регулярного оновлення навчальних даних, які з часом можуть втрачати актуальність, тому більш гнучким підходом у багатоканальній системі є керування поведінкою LLM через якісно сформований контекст і prompt-стратегії без зміни параметрів моделі. Такий підхід краще узгоджується з архітектурою керованого AI-компонента та дозволяє швидше адаптувати відповіді до змін сценаріїв і знань.

Додатково в роботі було сформульовано узагальнену модель процесу обробки повідомлень у багатоканальному середовищі, яка охоплює етапи отримання події з платформи, нормалізації даних, збереження історії взаємодії та формування відповіді.

Також встановлено, що централізоване керування діалогами через веб-інтерфейс є важливим практичним фактором для бізнес-застосувань, оскільки дозволяє об’єднати комунікацію з різних каналів, підвищити оперативність реагування та контролювати якість підтримки. Це створює передумови для подальшого розвитку системи в напрямі аналітики, автоматизації типових сценаріїв, розширення інтеграцій та впровадження більш гнучких механізмів персоналізації діалогів.

Отримані результати підтверджують, що запропонований підхід до інтеграції чат-ботів у веб-додатки забезпечує керованість, масштабованість та зручність експлуатації порівняно з автономними реалізаціями, а також створює практичну основу для якіснішої обробки діалогів за рахунок нормалізації даних і контрольованого підключення AI. Розроблений прототип може бути використаний як основа для подальшого розвитку системи, зокрема шляхом розширення підтримки інших платформ, посилення моніторингу та удосконалення механізмів оцінювання якості діалогів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Key concepts in the Bot Connector API. - Microsoft Learn. – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/bot-service /rest-api/bot-framework-rest-connector-concepts?view=azure-bot-service-4.0

2. What is Google Dialogflow, and How to Create a Chatbot? - GPTBots.ai – [Електронний ресурс]. – Режим доступу: https://www.gptbots.ai/blog/google-dialogflow

3. Singh, S. U., & Siami Namin, A. (2025). A survey on chatbots and large language models: Testing and evaluation techniques. Natural Language Processing Journal, 10, 100128.

4. Moataz Mohammed, Mostafa M. Aref (2022). Chatbot System Architecture. Computation and Language.

5. Polishchuk M., Savaryn P., & Furkalo S. (2022). Webhooks and Long Polling methods for hosting a Telegram bot. COMPUTER-INTEGRATED TECHNOLOGIES: EDUCATION, SCIENCE, PRODUCTION, (49), 86-92.

6. Dobbala, M. K., & Lingo lu, M. S. S. (2024). Conversational AI and Chatbots: Enhancing User Experience on Websites. American Journal of Computer Science and Technology, 7(3), 62–70.

7. Uriawan, W., Herdiyanto, R. F., Millah, R. I. S., Irhamnillah, S., & Gunawan, S. N. (2024). Real-Time Chatbot: Microservices Implementation in Distributed System Architecture. Preprints, 2024070028.

8. Singia A., Sukharevsky A., Yee L, Chui M., Hall B. (2025). The state of AI. How organizations are rewiring to capture value. QuantumBlack, AI by McKinsey.

9. AI chatbot: advantages and tips for companies - OTRS

10. Muhammad, T., & Stukalina, Y. (2025). The role of AI-powered chatbots in enhancing customer experience: Systematic literature review. Business Management, 2025, 211–217.

11. The 2023 State of Data + AI: How Businesses Are Preparing for the New Age of AI - Databricks. – [Електронний ресурс]. – Режим доступу: https://www.databricks.com/blog/2023-state-data-ai

12. Open-Source vs. Proprietary LLMs: 2025 Capability Guide - AlphaCorp AI. – [Електронний ресурс]. – Режим доступу: https://alphacorp.ai/open-source-vs-proprietary-llms-pros- cons-and-trends/

13. Emerging Architectures for LLM Applications. - Andreessen Horowitz. – [Електронний ресурс]. – Режим доступу: https://a16z.com/emerging-architectures-for-llm-applications/

14. Understanding the modern enterprise integration requirements. - Medium. – [Електронний ресурс]. – Режим доступу: https://medium.com/microservices-learning/understanding-the-modern-enterprise-integration-requirements-4ae58913a59d

15. Data Integration Layer Definition and Its Critical Role. - FanRuan. – [Електронний ресурс]. – Режим доступу: https://www.fanruan.com/en/blog/data-integration-layer-definition-critical-business-role

16. 11 Data Integration Strategies, Techniques, & Requirements. - ESTUARY. – [Електронний ресурс]. – Режим доступу: https://estuary.dev/blog/data-integration-strategy/

17. Data Integration Layer Definition - Apix Drive. – [Електронний ресурс]. – Режим доступу: https://apix-drive.com/en/blog/other/data-integration-layer-definition

18. Handling the Unique Challenges of Microservices Integration Testing - QA Madness. – [Електронний ресурс]. – Режим доступу: https://www.qamadness.com/handling-the-unique-challenges-of-microservices-integration-testing/

19. Microservices Architecture: Principles, Patterns, and Challenges for Scalable Systems. - Medium – [Електронний ресурс]. – Режим доступу: https://medium.com/@erickzanetti/microservices-architecture-principles-patterns-and-challenges-for-scalable-systems-9eac65b97b21

20. 10 Challenges to implementing Microservices - Fiorano – [Електронний ресурс]. – Режим доступу: https://www.fiorano.com/blogs/Ten_Challenges_to_implementing_Microservices

21. Advantages and disadvantages of microservices architecture - QA.com – [Електронний ресурс]. – Режим доступу: https://www.qa.com/resources/blog/microservices-architecture-challenge-advantage-drawback/

22. 10 Challenges In Implementing Microservices - Medium – [Електронний ресурс]. – Режим доступу: https://blog.bitsrc.io/microservice-challenges-146badd013e3

23. Pergantis P., Bemicha V., Drigas A. & Skianis Ch. (2024). AI Chatbots and Cognitive Control: Enhancing Executive Functions Through Chatbot Interactions: A Systematic Review.

24. Abulibdeh, A.; Zaidan, E.; Abulibdeh, R. Navigating the Confluence of Artificial Intelligence and Education for Sustainable Development in the Era of Industry 4.0: Challenges, Opportunities, and Ethical Dimensions. J. Clean. Prod. 2024, 437, 140527.

25. Bendig, E.; Erb, B.; Schulze-Thuesing, L.; Baumeister, H. The next Generation: Chatbots in Clinical Psychology and Psychotherapy to Foster Mental Health–A Scoping Review. Verhaltenstherapie 2022, 32, 1–13.

26. Kuhail, M.A.; Alturki, N.; Alramlawi, S.; Alhejori, K. Interacting with Educational Chatbots: A Systematic Review. Educ. Inf. Technol. 2023, 28, 973–1018.

27. Pergantis, P.; Bamicha, V.; Chaidi, I.; Drigas, A. Driving Under Cognitive Control: The Impact of Executive Functions in Driving. World Electr. Veh. J. 2024, 15, 474.

28. Chauncey, S.A.; McKenna, H.P. An Exploration of the Potential of Large Language Models to Enable Cognitive Flexibility in AI-Augmented Learning Environments. In Proceedings of the Future Technologies Conference; Lecture Notes in Networks and Systems; Springer: Cham, Switzerland, 2023; Volume 816, pp. 135–153.

29. Chauncey, S.A.; McKenna, H.P. Creativity and Innovation in Civic Spaces Supported by Cognitive Flexibility When Learning with AI Chatbots in Smart Cities. Urban Sci. 2024, 8, 16.

30. Parsakia, K. The effect of chatbots and AI on the self-efficacy, self-esteem, problem-solving and critical thinking of students. Health Nexus 2023, 1, 71–76.

31. Rostami, M.; Abadi, P.M. The Impact of Doing Assignments with Chatbots on The Students' Working Memory. Health Nexus 2023, 1, 64–70.

32. Chou, E.Y.; Hsu, W.C. Conversational Service Experiences in Chatbots: A Perspective on Cognitive Load. Manag. Rev. 2021, 40, 21–46.

33. Fabio, R.A.; Plebe, A.; Suriano, R. AI-Based Chatbot Interactions and Critical Thinking Skills: An Exploratory Study. Curr. Psychol. 2024, 1–14.

34. Baha, T.A.; El Hajji, M.; Es-Saady, Y.; Fadili, H. The Power of Personalization: A Systematic Review of Personality-Adaptive Chatbots. SN Comput. Sci. 2023, 4, 661.

35. RabbitMQ Documentation. – RabbitMQ. – [Електронний ресурс]. – Режим доступу: https://www.rabbitmq.com/documentation.html

36. Telegram Bot API. – Telegram. – [Електронний ресурс]. – Режим доступу: https://core.telegram.org/bots/api

37. Docker Documentation. – Docker. – [Електронний ресурс]. – Режим доступу: https://docs.docker.com/

38. FastAPI Documentation. – FastAPI. – [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/

39. ReactJS Documentation. – ReactJS – [Електронний ресурс]. – Режим доступу: https://react.dev/

40. SQLAlchemy 2.0 Documentation. – SQLAlchemy – [Електронний ресурс]. – Режим доступу: https://docs.sqlalchemy.org/en/20/